1989

Design and development of computer aided data acquisition environments for anemometers.

Alex. Tsui
University of Windsor

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

DESIGN AND DEVELOPMENT OF

COMPUTER AIDED DATA ACQUISITION ENVIRONMENTS FOR ANEMOMETERS

by

© Alex Tsui

A Thesis
Submitted to the Faculty of Graduate Studies through the
Department of Mechanical Engineering in Partial Fulfilment
of the Requirements for the Degree of
Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada

1988

ISBN 0-315-50540-0

ABSTRACT

Computer Aided Data Acquisition (CADA) environments for the Constant Temperature Anemometer (CTA) and the Laser Doppler Anemometer (LDA) systems manufactured by Dantec Electronics have been developed. These environments (CADA-CTA and CADA-LDA) reduce the tedious manual process of traversing the sensor and recording the data; are generic, easy to use, flexible, accessible and affordable. The IBM Personal Computer (IBM-PC) is chosen as the controlling computer because it is one of the most popular and it is found in most laboratories. The software developed in this thesis makes use of an IBM-PC to control and acquired data from CTA, Signal Analysis Equipment (SAE), LDA, sensors and devices. These environments have been used in research projects and the results indicate that the hardware and software discussed in this thesis are very useful.

ACKNOWLEDGEMENTS

The author is grateful to Dr. K. Sridhar and Dr. G. W. Rankin for their interest and continuous support.

Special thanks are also due to Dr. R. G. S. Gaspar and Dr. J. Soltis for their invaluable help in this project.

The author is deeply indebted to the Mr. R. Tattersall, Mr. W. Beck and J. Novosad for their technical support.

The author wishes to thank his wife, Athena for her ever present encouragement and patience throughout his study.

The author also wishes to thank his parent and family for their encouragement and support.

The financial support received from the Natural Sciences and Engineering Research Council, Canada through Grant Numbers A-2190 and A-1403 is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES AND ILLUSTRATIONS

LIST OF TABLES

NOMENCLATURE

δ_f	- distance between Doppler fringes	(m)
A	- area	(m^2)
d	- diameter	(m)
f_D	- Doppler frequency	(Hz)
H	- heat transfer rate	(W)
k	- thermal conductivity	(W/m·K)
Nu	- Nusselt number	
θ	- angle	(radian)
Pr	- Prandtl number	
Re	- Reynolds number	
t	- time	(s)
T_f	- fluid temperature	(K)
T_w	- hot wire temperature	(K)
U	- velocity	(m/s)
V	- voltage	(volt)

CHAPTER 1 - INTRODUCTION

There is a variety of data acquisition equipment available for fluid mechanic measurements, such as velocity, pressure, temperature, ...etc. However, the data collection involved in an accurate experiment is very time consuming and tedious. For a thorough experiment, it may take hours, months or even a longer period to accomplish data acquisition. This is especially true when turbulent flow phenomena are being studied. Therefore, tools are needed to aid the data acquisition process. In addition, these tools must be user friendly, generic, flexible, accessible and affordable for general use. The purpose of this project is to develop data acquisition environments for acquiring data from two anemometer systems used in the Mechanical Engineering laboratory. They are the Constant Temperature Anemometer (CTA) and the Laser Doppler Anemometer (LDA). An IBM personal computer is used to control and monitor the anemometers and various other equipments. The CTA system communicates with the IBM-PC computer via a standard IEEE-488 interface. In the case of the LDA, a hardware interface card has been specially developed. Refer to Fig. 1.1 and Fig. 1.2 for their respective configuration. In order to make these environments user friendly and generic, an interactive windowing-user interface and a simple interpreting language have been developed. The environments are sufficiently flexible so as to allow other electrical and electronic controlled devices to be added. The control of a four axis traversing unit has been implemented for sensor positioning. A stepper motor control capability has been added for the

control of sensor rotation for the CTA environment.

The remainder of the thesis has been divided as follows. General descriptions of the Constant Temperature and Laser Doppler Anemometer Systems will be discussed in Chapter 2. The computer aided data acquisition user environment is described in chapter 3. It includes the basic concept and the commands that are available. Examples and demonstration programs are also included in this chapter. Chapter 4 is an explanation of the hardware interface for the LDA counter processor. It is subdivided into three sections. The first section describes the hardware requirement of the Dantec Laser Doppler Anemometer and the design of the PC-LDA interfacing system. The second section discusses the various methods of data acquisition and their differences. The last section describes the PC-LDA interface card. A discussion and recommendations are presented in Chapter 5.

Appendices A to D contain source codes and help files for the two computer aided acquisition programs. Appendix E contains a listing of the programs which demonstrate the three modes of data acquisition which the PC-LDA interface is capable of, and also a program which demonstrates the procedures used to initialize the PC-LDA hardware. Lastly, circuit diagrams of the PC-LDA can be found in Appendix F.

CHAPTER 2 - THEORY OF TWO ANEMOMETER SYSTEMS

2.1 - Constant Temperature Anemometer (CTA)

The Constant Temperature Anemometer (CTA) uses a hot-wire as the sensor. It is one of the most important tools in measuring the variables such as the components of the mean and the velocity fluctuation which occur in turbulent flows.

The hot-wire anemometer consists of a very fine short metal wire, which is heated by an electrical current. The wire temperature is lowered by convective heat transfer to the flowing fluid, and consequently, the electrical resistance of the wire is reduced. For turbulence measurements in gases, a wire of 2 to 10 microns in diameter and 1 to 2 mm in length is used. Materials such as platinum, platinum-iridium, and tungsten are usually used.

When the hot-wire is introduced in a flow, heat is transferred out of the wire by radiation and conduction along the wire to its end supports, natural or free convection, and forced convection by the fluid. The radiative heat loss is typically 0.1 percent of the electrical input and can be neglected. Free convection is only important for very low speeds. For flow speeds, which are greater than 5 cm/sec, it can be neglected. Therefore, the main contributions to heat transfer from the wire are conduction to the supports and forced convection to the fluid flow. For the conduction to the support, the supports are much thicker than the wire for strength reasons and so that they will not be heated appreciably by the electrical current. The ratio of the total conductive heat transfer to that convected to the fluid is approximately 15 percent

for a typical tungsten wire. For the forced convection, heat transfer from hot wires is usually expressed in terms of the Nusselt number, which is equal to the rate of heat transfer to the fluid per unit area, H/A, divided by the product of the thermal conductivity of the fluid, k and a typical temperature gradient. Taking the typical temperature gradient as (Tw-Tf)/d the Nusselt number becomes

$$Nu = \frac{H / A}{k * (T_w - T_f) / d}$$

The Nusselt number is a function of Reynolds number, Re, and the Prandtl number, Pr,

$$Nu = f(Re, Pr)$$

King's semi-empirical formula is the most widely accepted relationship and is given as

$$Nu = A + B * Re^{0.5}$$ where A and B are constants.

Hot-wire anemometers are divided into two types.

1) In a Constant Current Anemometer (CCA), the current flowing through the wire is kept constant and the variation of temperature (or resistance) is monitored.

2) In a Constant Temperature Anemometer, the temperature (or, the electrical resistance) is maintained constant by varying the current through the wire. The variation of this current or voltage is taken as the measurement of the fluid velocity.

Since the Constant Temperature Anemometer is to be implemented in the present work, the operation of the Constant Current Anemometer is not discussed here.

Operation of the Constant Temperature Anemometer: The hot-wire is

placed in one arm of a Wheatstone bridge with a fixed resistor placed in the opposite bridge arm for defining the overheat ratio and hence the temperature of the hot-wire. A feed back amplifier regulates the bridge current in order to maintain the bridge balance in all situations, that is, the hot-wire resistance (therefore it's temperature) is kept constant and independent of the cooling rate. For continuum flow, it has the following expression, normally referred to as King's law.

$$E^2 = A + B * U^n$$

The constants A and B do not only depend upon the dimensions and operating conditions of the hot-wire but upon the physical properties of the gas and on the flow conditions as well. The exponent n is constant only in a limited range of velocity. Figure 2.1 shows the functional block diagram of a CTA unit [1].

2.1.1 - Components of The CTA System

The following sections describe the major components of a Constant Temperature Anemometer system.

2.1.1.1 - Linearizer

The output voltage from a constant temperature anemometer is a non-linear function of the velocity of the fluid passing over the probe. Although it is inconvenient, it is possible to correct this non-linearity in measurements of laminar flows or in flows with low degree of turbulence. However, in highly turbulent flow, the distortions of the signal which occur are difficult or impossible to correct. It is thus convenient and often necessary to linearize the CTA signals. The

linearizer is placed in the signal path, and its transfer function is adjusted to be the inverse of the transfer function of the anemometer. Therefore, the distortions of the velocity information are eliminated.

Principle and function of the linearizer: The input signal is amplified after a zero offset. Therefore, the input signal V_B is transferred to the normalized signal X. When the velocity varies from 0 to U_{max}, the input signal V_B changes from V_0 to V_{max}. Values of X, from 0 to 10 volts, correspond to this. The rest of the block diagram shows the linearizing circuit. The circuit implements the mathematical expression:

$$Y = 10^{(A + BX + EY)} + CX + D$$

where A, B, C, D and E are constants and Y is the linearized output.

2.1.1.2 - Signal Conditioner

This unit is designed to amplify and filter the output signals from the CTA bridge or linearizer. Normally, the signals are amplified to a level which is required for further processing in the Signal Analysis Equipment (SAE). It features:

1) High pass and low pass filtering.

2) Amplification (Gain).

2.1.1.3 - Signal Analysis Equipment

The signal analysis equipment consists of two units: the MEAN VALUE unit and the RMS unit.

The mean value unit is a module which is used to get the mean voltage from the CTA unit.

The main function of the RMS unit is to measure the RMS value and the correlation of the signals. In the RMS mode, the result displayed indicates the value of:

$$\text{DISPLAY RESULT} = \text{SQRT} \left(V_{\text{channel a}} * V_{\text{channel b}} \right)$$

In the CORR mode, the result shown on the display indicates the value of:

$$\text{DISPLAY RESULT} = \left(V_{\text{channel a}} * V_{\text{channel b}} \right)$$

2.2 - Laser Doppler Anemometer (LDA)

The LDA employs an optical method which offers a non-intrusive means of velocity measurement . It responds rapidly and is suitable for measuring high frequency turbulence fluctuations. It indicates the velocity of a fluid flow by means of a counting technique. Fundamentally, the LDA counter processor measures the time for a particles presented or seeded in the fluid to cross light fringes located within the measuring volume, thus enabling the fluid velocity to be evaluated.

A schematic of the LDA is shown in Fig. 2.2. This is called the dual-beam method. The laser beam is split into two equal intensity beams outside the test section through a beam splitter. Then, both of the beams focus with a focusing lens at the measurement volume. The photo multiplier monitors any moving particle which passes through the beam crossing, and detects the Doppler shift in frequency which is directly proportional to the flow velocity. The aperture is a shield for non-coherent scattered light as well as background light. A Helium-Neon, He-Ne, gas laser is usually used in LDA work. The He-Ne laser operates at a wavelength of 632.8 nm with a bandwidth of about 10 Hz. The LDA counter processor measures the velocity of the scattering particles. If they are

sufficiently small, the slip velocity between particles and fluid will be small and thus an accurate indication of fluid velocity will be obtained.

From the geometry of the optical beams and the wavelength of the laser light, the distance between the fringes is known. The light scattered from the particle is detected, and the result is a frequency modulated current burst from the detector. The two intersecting laser beams form an interference or fringe pattern at the intersection as shown in Fig. 2.3. The Doppler frequency (f_D) is the frequency of the burst:

$$f_D = 2 * U_x / \sin (\Theta / 2)$$

where Θ : Angle between the Laser beams.

U_x : Velocity of the particle.

By measuring the time for a certain number of zero crossings of the amplified and filtered detector signal, the Doppler burst frequency is detected. Refer to Fig. 2.4 for a illustrated explanation.

The LDA counter processor is basically an electronic stop watch. It measures the time it takes for a particle to cross a known number of interference fringes in the measuring volume (refer to Fig. 2.4).

2.2.1 Dantec LDA Counter Processor Modes

There are three modes of operation of LDA counter processor which correspond to three methods of determining the doppler frequency.

1) **Fixed Fringe (Mode 1)**

With this mode the counter processor will calculate the Doppler frequency and sample interval generated by the moving particle using a total of 8 fringes of the detected Doppler signal.

Doppler frequency = 8 / (time for 8 fringes)

This mode is used to measure flow when many particles are in the measuring volume at one time. This will produce an output for every 8 fringes and this mode will yield results in the case of poor signal to noise ratios.

This mode supports the 5/8 data validation method. This method is use to check for the valid data. Basically, the counter processor times the first 5 and 8 cycles and determines the difference between the two Doppler periods. If the difference is within a selected tolerance, the sample is considered to be good.

2) Variable Fringe (Mode 3)

This mode is used to measure a single burst of data. It will generate the total burst time, sample interval and fringe count for analysis. One can apply the following formula to calculate the Doppler frequency :

Doppler frequency = (fringe count) / (burst time)

With this mode one can achieve one set of data per burst.

3) Combined (Mode 2)

This mode is a combination of modes 1 and 3. It determines Doppler frequency from the first 8 fringes of the detected Doppler signal burst. The sample interval and fringe count are obtained from the complete burst. This mode is used for single burst measurement or one measurement per burst situation, and it requires a good signal to noise condition and low seeding rates. The Doppler frequency formula is as follow :

Doppler frequency = (fringe count) / (burst time)

and

Doppler frequency = 8 / (time of 8 fringes)

This mode also supports the 5/8 data validation method.

4) **Transit (Mode 4)**

This is the dual focus mode. With this mode one can obtain the particle transit time, and sample interval. This mode does not employ the Doppler effect. Refer to DANTEC's LDA Counter Processor Instruction Manual [9] for more detail.

CHAPTER 3 - DESCRIPTION OF THE CADA USER ENVIRONMENT

3.1 - Basic Concept of the CADA user interface

The two environments that have been developed are called 'CADA-LDA' and 'CADA-CTA', where 'CADA' stands for Computer Aided Data Acquisition and 'LDA' and 'CTA' identify which environment the packages are designed for. Therefore, 'CADA-LDA' implies Computer Aided Data Acquisition for the Laser Doppler Anemometer system.

Effort has been made to make sure that these software environments are easy to learn and use. Interactive windowing environments were developed to produce a simple user interface. The user enters commands via the input window while the computer sends its response via the device and user windows. A command interpreter has been developed to allow the user to control devices which in turn acquire data for later analysis. The command file includes facilities to allow the user to write simple programs. These programs can either be entered interactively or saved as a standard ASCII text file called a CADA command file. By calling a command file within the CADA environment, one can perform automated or unattended data acquisition. One facility of this command language allows the user to store the acquired data in an ASCII or text file, and one can a execute a DOS program within the 'CADA' environment to analyze the data. These programs can be custom or commercial programs.

The following section will explain the interactive windowing environment of the user interface and other sections contain an overview of all the commands which are implemented in the 'CADA' environment.

3.1.1 - Interactive Windowing Environment

The computer screen is divided into multiple display windows (see Fig. 3.1). The format of this screen may vary since the location of each of the windows can be rearranged by the user. Each window has specific purposes, but the most important windows are: 1) input window, 2) user window, 3) help window, 4) device windows, and 5) time and date window. The detailed explanation of the purposes and the usefulness of each of the windows is presented later in this section. This user interactive environment also accepts a few function keys such as F1 and F2 to allow the user to access some useful facilities easier and faster. When F1 is pressed, a help screen will appear and the user can scroll about to locate the desired help information. Pressing F2 allows the user to rearrange each of the windows' location and size. A scrolling feature is also available to allow the user to scroll the input, user, help or other windows to review previous commands or error messages.

The purpose of the Input window is to allow the user to enter commands for the CADA program to interpret. The User window is provides output only. This window displays response messages from the CADA software and is designed mainly for debugging the user's commands. The help window can be accessed by either typing in the "Help;" command or via the "F1" key. Through this window, one can scroll up and down a help file to learn this package. The device windows are used to inform the user of the device status. An example is an indication of the location of the sensor or the settings of the CTA devices. The time and date window displays the time and the date plus the availability of memory in the PC system.

3.1.2 - Command Interpretive Language

This environment is controlled by a command interpretive language. This language is very limited but much easier to learn and use than BASIC. Each command is composed of a primary command followed by a colon ':' separator, then one or more sub-commands or secondary tokens. At the end of the statement, a semi-colon ';' is required as the statement terminator.

Each sub-command may have an option to assign a value to it, in such a case, an equal sign '=' must be inserted between the sub-command and its value. If multiple sub-commands are used, commas ',' are required to separate them.

This interpretive language is similar to PC-DOS BATCH language. One difference between this environment and the BATCH environment is that the user must enter the statement in the input window, and any message generated by this command will be displayed in the user and device window.

A program (CADA command program) can also be written to do automatic data acquisition. There is a full set of commands which allow the user or programmer to perform looping, data file handling, waiting and execution of other DOS programs.

3.2 - Command Description

This section gives an overview description of all the commands available in the two data acquisition packages. A detailed explanation of each command can be found in Appendices A and C.

3.2.1 - DOS Command Syntax

The syntax used to invoke this program in the DOS environment is as follows:

SYNTAX: CADA-xxx [CADA-xxx command line]

where xxx is either 'CTA' or 'LDA', this defines which anemometer is to be used.

An optional command line may be included in the invocation of this program. This optional command line will be treated as the first line to be interpreted by the program. If this option is not included, the program will wait for user input after the initialization.

Example:

To invoke the CTA environment and let the program prompt for input command.

Type >> CADA-CTA

The following syntax can be used to invoke the LDA environment and execute a command file to do automatic data acquisition. Note that "acquire.cmd" is the name of a CADA command file written by the user with an ASCII editor.

TYPE >> CADA-LDA execute:file=acquire.cmd;

3.2.2 - Interpretive Command Syntax

```
Primary_Cmd : Sub_Cmd = Value [, Sub_Cmd = Value] ;
***         *   ***   *  ***   *   ***   *  ***   *
```

*** -> Primary or Secondary/Option command.

* -> Separators.

[ ] -> Option.

Don't forget the separators. They are essential !

All Primary commands must end with a colon ----------> ' : '.

The secondary command separator is a comma ----------> ' , '.

All command lines must terminate with a semi-colon --> ' ; '.

All strings after ';' will be considered as comments.

3.2.3 - Parameter Definition

1) Number Definition

n2= Integer values in the range of 0 to 2 (e.g., 0,1,2)

n7= Integer values in the range of 0 to 7 (e.g., 0,1, .. 7)

r = Real number (e.g., 1.128)

Ir= Incremental value (e.g., i+1.2 or i-0.5)

2) Optional Parameter Definition [ ]

Note that tokens within a set of square brackets are used to

indicate that one or none of the specified options can be selected. Therefore, if [token1 or token2 or token3] is given, one or none can be chosen from that list.

3) Significant Characters of a Command

Only the first three characters of the token or command are important.

4) Blanks

Blanks are neither token separators nor of any importance except when performing a DOS command.

5) Escape Key

The escape key '<ESC>' can be used to terminate an executing command or command file.

3.2.4 - Facilities Available for both CTA-SAE and LDA

The following sections outline the operations provided both CTA and LDA environments.

3.2.4.1 - System Commands

1) Help ;

Same as F1. The help file will appear on the Help screen. One can control the movement of the window with the cursor keys. In order to exit this file, press F10.

2) Quit ;

This command will close all opened files and return to DOS.

3) Window ;

Same as F2. It will allow the user to use the cursor keys to change

the window setup. Key F5 will scroll the contents within that window. Key F6 can change the size of the window. Key F7 allows the user to move the current window to any location within the screen. Key F8 will change to a different window. Key F9 is to save, recall or delete the window configuration file. Key F10 is to exit the window setup menu.

3.2.4.2 - Looping and Waiting Commands

1) DO : n9, Times = n32767 ;
.
.
END n9 ;

The label of each DO command must match with an END command of the same label.

2) Wait : Key,
Time = hh:mm:ss,
Delay = r ;

The program will wait for the key pressed when "key" is specified, the exact time when "Time=hh:mm:ss" is specified or r second delay when "Delay= r " is defined. The user can specify multiple sub-commands and it will be executed in sequence.

3.2.4.3 - Command File Execution Command

Execute : File = drive:path\filename.ext ;

One can write their own controlling command file with his choice of editor and execute them within the CADA program. It allows the user to perform an automatic data acquisition program.

3.2.4.4 - File Handling Command

File : Open = <filename.ext> , To_File_Number = n9 ;

File : Close , To_File_Number = n9 ;

File : Store , To_File_Number = n9 ;

File : Format = (options) , To_File_Number = n9 ;

where options can be any combination of the following separated by a comma

Time : generate the current time

Date : generate the current date

Sp = n : generate n spaces

The above commands can be used to open a data file, close a data file, define the writing format and store data to file number <n9>. Moreover, the format of the data file is set by the "File : Format=(option), .." command, where "option" defines the output format. Refer to section 3.2.5.5 and 3.2.6.5 for the available options in each of the CADA environments.

3.2.4.5 - DOS File Execution Command

1) DOS [-p] ;

This command will open a subprocess to DOS. Type EXIT to return to CADA program. If the "-p" option is given, CADA will bypass the question "press RETURN to continue....".

2) DOS [-p] := < drive:path\filename [parameters] >

This command will open a DOS process and execute the given DOS command. This DOS command can be a DOS SHELL (i.e., dir, copy), a '.COM' command program, an '.EXE' execution program or a '.BAT' batch file.

3.2.4.6 - Four Axis Traversing Device

```
Enable  : Traver ;
Disable : Traver ;
Read    : Traver ;
Traver  : = INIT ;
Traver  : X = [r | Ir], Y = [r | Ir], Z = [r | Ir], R = [r | Ir],
          Delay = [ r | Wait | All_Finish ] ,
          Device_number = n15 ;
```

The Enable command will enable the program to talk to the traverse and the Disable command will have the opposite effect. The Read command can be used to read the position of the traverse. The INIT sub-command is used to zero the position counters. If the Delay=Wait is used, the software will only wait for the last specified axis to become stable. If Delay equals r, it will delay for r second(s). If Delay=All_Finish is used, it will wait until all axes' movements become stable. X, Y and Z are the three axes. There are two methods to set the X,Y,Z. One can either set it by absolute position, r or by relative position, Ir. If r is given as say 2.5, the traverse will move to location 2.5 with respect to the traverse zero. If Ir is say I+4.2 the traverse will move 4.2 units in the positive direction from the last stable location. The IEEE-488 device number is set by the last sub-command.

3.2.5 - Facilities Available for the CTA-SAE Environment Only

The following sections outline the operations provided exclusively for the CTA-SAE environment.

3.2.5.1 - Commands Used to Enable, Disable and Read Devices

```
Enable  : CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, PROBE, All;
Disable : CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, PROBE, All;
Read    : CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, PROBE, All;
```

The three commands are used to enable, disable and read information from the devices which correspond to the above sub-commands. The devices which correspond to each command are described below. However, if "ALL" is specified for the option, the LDA, AINPUT and AOUTPUT will be selected.

3.2.5.2 - Constant Temperature Anemometer Device Controlling Commands

```
1)  CRMS : CHA          = n7,     - 1, 2, ... 6
           TC           = r,      - 0.1, 1, 10, 100 sec.
           DEVICE_NUM   = n15 ;   - 0, 1, ... 15
```

The CTA RMS unit (CRMS ; Dantec part # 56N10 [2]) is a true RMS voltmeter which is intended for measuring AC-components of the 56N20 Signal Conditioner output signal. CHA is used to set the input channel number. TC is used to set the time constant. The IEEE-488 device number is set by the last sub-command.

```
2)  CMEAN : CHA          = n7,     - 1, 2, ... 14
                                     15 =  0.0  volt ref.
                                     16 = 10.0 volts ref.
            TC           = r,      - 0.1, 1, 10, 100 sec.
            DEVICE_NUM   = n15 ;   - 0, 1, ... 15
```

The primary purpose of the CTA Mean Value unit (CMEAN ; Dantec part # 56N11 [3]) is to measure the DC component of the output signal from the other modules in the multichannel CTA system. CHA is used to set the

input channel number. TC is used to set the time constant. The IEEE-488 device number is set by the last sub-command.

```
3)   COND  : CHA          = n2,            - 0, 1, ... 7
             GAIN         = r,             - 1.0 ... 500 times
             LP           = r,             - 0.1 ... 300 kHz.
             HP           = r,             - 0.1 ... 300 Hz.
             DISPLAY      = [ON | OFF] ;   - ON or OFF
             DEVICE_NUM   = n15,           - 0, 1, ... 15
```

The CTA Signal Conditioner Unit (COND ; Dantec part # 56N20 [4]) is used to improve the CTA signal by allowing the user to control the amount of gain to the input signal and applying high pass and low pass filtering.

The signal conditioner unit (COND) has an extra sub-command called "DISPLAY". With this option, one can enable or disable the displaying of high-pass filter setting, HP, low-pass filter setting, LP and amplifier gain setting, GAIN, to the selected channel after either a READ or a COND command is executed. The IEEE-488 device number is set by the last sub-command.

3.2.5.3 - Signal Analysis Equipment (SAE) Device Controlling Commands

```
1)   SRMS  : CHA          = n7,            - 0, 1, ... 7
             CHB          = n7,            - 0, 1, ... 7
             TC           = r,             - 0.1, 1, 10, 100 sec.
             DEVICE_NUM   = n15 ;          - 0, 1, ... 15
```

The SAE RMS unit (SRMS ; Dantec part # 56N25 [7]) is a true RMS voltmeter which is intended for measuring AC-components of the 56N20 Signal Conditioner output signal. CHA and CHB are used to set the input channel number. TC is used to set the time constant. The IEEE-488 device number is set by the last sub-command.

```
2)   SMEAN : CHA          = n7,          - 0, 1, ... 7
             TC           = r,           - 1.049 ... 1049 sec
             DEVICE_NUM   = n15 ;        - 0, 1, ... 15
```

The SAE Mean Value unit (SMEAN ; Dantec part # 56N22 [6]) is to measure the DC component of the output signal from the other modules in the multichannel SAE system. CHA is used to set the input channel number. TC is used to set the time constant. The IEEE-488 device number is set by the last sub-command.

```
3)   SAM   : TRIGGER        = n3,          - 0,  1, ...3
             PROCESS        = n31,         - 0,  1, ...31
             ACCTIM         = r,           - 0.00025...687194 sec.
             TEST           = [0 | 1],     - for H/W testing
             DEVICE_NUM     = n15,         - 0,  1, ...   15
```

The Signal Analysis Module (SAM : Dantec part # 56N30 [8]) inserts into an SAE main frame and operates under software control. It allows simultaneous measurement on eight input signals. These may be eight signals from the AC-bus or eight signals from the DC-bus. TRIGGER is used to specify the mode of triggering. PROCESS defines modes of operation, channels to be used and physical units to be used when results are later read out from the unit. ACCTIM is the integration time. TEST is for hardware testing. Refer to Dantec's Signal Analysis Module Instruction Manual [8] for detail description of the above option. The IEEE-488 device number is set by the last sub-command.

```
4)   SCORR : CHX            = n7,          - 0, 1, ... 7
             CHY            = n7,          - 0, 1, ... 7
             IDELAY         = r,           - Initial delay in sec.
             SDELAY         = r,           - Step delay in sec.
             TFACTOR        = n,           - Time factor
```

DEVICE_NUM = n15 ; - 0, 1, ... 15

The SAE Correlator unit (SCORR; Dantec part # 56N12 [5]) is a plug-in module for the SAE system. In conjunction with a SAM, the unit forms a correlator capable of providing 8 simultaneous values of a delayed auto- or cross-covariance function. CHX and CHY are input channels. IDELAY is the initial delay in second. SDELAY is the step delay in seconds. TFACTOR is the time factor. Refer to Dantec's Correlator (56N12) Instruction Manual [5] for a detailed explanation of the above options. The IEEE-488 device number is set by the last sub-command.

3.2.5.4 - Probe Driver Controlling Command

PROBE : = INIT ;
PROBE : ANGLE = [r | Ir] ;

This command is used to rotate the hot-wire probe which is mounted on a special stepper motor mechanism. The INIT sub-command can be used to zero the angle counter. The ANGLE sub-command can be used to rotate the probe to an absolute, r, or and relative, Ir, angular position.

3.2.5.5 - Data file options peculiar to CTA-SAE

FILE : Format = (options), to_file_# = n9 ;

Options can be any combination of the following separated by a comma:

```
COND = channel_#          : produce data from the Signal Conditioner
CRMS  or  CRMS  = data    : CTA RMS device information
CMEAN or  CMEAN = data    : CTA MEAN value device information
SRMS  or  SRMS  = data    : SAE RMS device information
SMEAN or  SMEAN = data    : SAE MEAN value device information
SCORR or  SCORR = data    : SAE Correlation unit information
SAM   or  SAM   = data    : SAE Signal Analysis Module's information
Probe                     : Angle setting of the probe rotation unit
Traverse                  : X, Y, Z position of the traversing unit
```

Note, if the "= data" option is specified, the "File: Store" command will only generate the device output data and will not generate any device setting information.

3.2.6 - Facilities Available for the LDA Environment Only

The following sections outline the operations provided exclusively for the LDA environment.

3.2.6.1 - Commands Used to Enable, Disable and Read Devices

ENABLE : LDA, AINPUT, AOUTPUT, ALL ;

DISABLE : LDA, AINPUT, AOUTPUT, ALL ;

READ : LDA, AINPUT, AOUTPUT, ALL ;

The three commands are used to enable, disable and read the above sub-commands. If "ALL" is specified for the option, the LDA, AINPUT and AOUTPUT will be selected.

3.2.6.2 - LDA Counter Processor Programming Command

[Primary]	[Secondary Token and default setting]	
LDA:	Mode	= 0,
	Nsets	= 0,
	Interval time	= 0,
	Display Mode	= ON,
	DMA channel #	= 3,
	Time Base	= 2,
	Delay Factor	= 1.00;

The followings are the description of the Secondary Tokens:

1) MODE = n4,

All modes yield the sample interval time.

mode 0 : Raw data of P, F, T and D. Refer to section 4.1.1 for description.

mode 1 : Fixed mode yields Doppler frequency.

mode 2 : Combined mode yields Doppler frequency and fringe count.

Mode 3 : Burst mode yields burst time and fringe count.

mode 4 : Transit mode yields transit time.

2) NSETS = n10000,

Number of consecutive readings (accept range from 0 to 10,000).

3) INTERVAL_TIME = r,

Approximate time delay between each consecutive reading (accept range from 0.05 sec to 32.0 sec). If it is set to 0.0, the DMA mode will be activated. This mode may acquire data as fast as 500 kbyte/sec or approximately 83 thousand sets of data per second which depends on the signal data rate.

4) DISPLAY_MODE = [ON or OFF or AGAIN],

ON : Display results in the data window when a "READ : LDA" command is send.

OFF : Suspend data display after reading.

AGAIN : Redisplay the LDA results on the data window.

5) DMA_CHANNEL# = [1 or 3],

The value depends on the PC-LDA hardware configuration. This allows one to set the actual DMA channel used by the PC-LDA interface card.

6) TIME_BASE = n3,

This selects the time base for the sample interval calculation.

0 : time base = 1. (external time base used).

1 : time base = 0. (not allowed).

2 : time base = 1e-3 sec.

3 : time base = 100e-9 sec.

7) DELAY_FACTOR = r,

This factor is used to generate the actual delay between each consecutive reading when the interval time is set to non-zero. If the default setting is 1.00, it is good for running on the standard 4.77 Mhz IBM-PC, and may be required to be changed if other compatible systems are used. Refer to Dantec's 55L90a LDA Counter processor Instruction manual [9] for detail description.

Example 1:

To program the CADA-LDA software to receive 20 sets of readings as fast as the LDA counter processor can generate using mode 1 (FIXED MODE), and to set the sampling interval time base to 1 msec. One should type in:

```
CMD > LDA : NSETS = 20, INTERVAL_TIME = 0.0, TIME_BASE =2;
CMD > READ: LDA;
```

The "READ : LDA" command will read the data from the LDA counter processor.

If one wishes to reprogram a particular setting, one must type in a command similar to the one above with the new information. For

example, if one wants to change the sampling rate to 5 readings per second, the following must to be written.

```
CMD > LDA : INTERVAL_TIME = 0.2;
CMD > READ: LDA;
```

To redisplay the data, just read. One must type in:

```
CMD > LDA : DISPLAY = AGAIN;
```

Example 2:

If one is using an IBM-AT to run this software, it may be required to change the DELAY_FACTOR to generate a desirable delay between consecutive readings (with respect to a particular INTERVAL_TIME setting). To determine such DELAY_FACTOR, one may follow this example.

1 - Assume the interested INTERVAL_TIME is 0.1 sec.

2 - Configure the LDA to read 600 sets of readings, with DELAY_FACTOR set to 1.0.
(note: if the DELAY_FACTOR is correct, this should take 60 sec. (theoretical time))

3 - Measure the actual time for taking these 600 sets of readings.

4 - The new DELAY_FACTOR for this particular example will be DELAY_FACTOR = (theoretical time) / (actual time).

```
CMD > LDA : NSETS=600, INTERVAL_TIME=0.1,
            DELAY_FACTOR=1.0;
CMD > ;
CMD > READ: LDA; ... start stop watch ...
```

```
CMD > ;
CMD > ;          ... new DF = theoretical time / actual time
CMD > ;
CMD > LDA : DELAY_FACTOR = <new DF>;
```

3.2.6.3 - Analog Input Commands

This command is designed to scan a sequence of (twelve-bit Analog to Digital Converter manufactured by ISAAC [20]) channels in ascending numerical order. LOW and HIGH identify the channel to be scanned and the sequence. COUNT defines how many sets of readings are taken, RATE defines how fast to sample each channel, while the OPTION command is available to change the ISAAC default settings. The DISPLAY command controls the display of the result after a read operation.

[Primary]	[Secondary Token]	
Ainput:	LOW CHANNEL= n15,	- First channel.
	HIGH CHANNEL= n15,	- Last channel.
	COUNT = n32767,	- Number of consecutive readings.
	RATE = n32767,	- Number of samples per second.
	OPTION = <string>,	- Refer to ISAAC reference menu.
	DISPLAY = [ON | OFF | AGAIN];	- Display after read

1) LOW = n15,

It indicates the starting channel.

2) HIGH = n15,

It indicates the ending channel. If LOW and HIGH are equal, it will scan only one channel.

3) COUNT = n32767,

This sub-command indicates how many times to scan the channels. The

total number of samples input is the number of channels scanned times the count.

4) RATE = n32767,

This is used to select the scanning rate. This parameter controls the rate at which each sequence is sampled. Within each set of channels the inter-channel time is as short as possible. RATE determines the rate at which the complete scans are executed. If RATE equals zero, external clocking will be used. Refer to the ISAAC documentation for information [20].

5) OPTION = <string>,

The valid optional parameters are the standard DEVICE, UNIT, and CONTROL parameters, the EXECUTION mode parameter, the STORAGE parameter, and the INITSOFF flag. Refer to page 4-2 in the ISAAC documentation [20] for detailed information.

6) Display Mode = [ON or OFF or AGAIN],

If "ON" is used, it will display results in the data window after a "READ : LDA" command is sent. If "OFF" is used, it will suspend data display after reading. If "AGAIN" is used, it will redisplay the LDA results on the data window.

Example:

To scan 10 sets of analog input data starting at channel 2, and ending at channel 5, issue the following.

ex: CMD > Ainput: low = 2, high = 5, count = 10;

3.2.6.4 - Analog Output Command

This command takes a digital value and outputs it to the specified twelve-bit digital to analog converter channel on the ISSAC Analog Module [20].

[Primary] [Secondary Token]

Aoutput:	CHANNEL #	= n15,	- Analog output channel.
	DAC	= n4095,	- Digit to Analog value.
	OPTION	= <string>,	- Refer to ISAAC reference menu.

1) CHANNEL # = n15,

This selects the analog output channel.

2) DAC = n4095,

This is the digital value the ISAAC system used to convert to an analog voltage. A zero implies -5 volts. DAC equals 4095 implies +5 volts.

3) OPTIONS = <string>,

The valid optional parameters are the standard DEVICE, UNIT, and CONTROL parameters, the EXECUTION mode parameter, the STORAGE parameter, and the INTSOFF flag. Refer to page 4-2 in the ISAAC documentation [20] for detail information.

Example:

To set analog output channel 3 to 3.5 volts, the 3.5 volts constant must be converted to a digital representation and the following is required:

since 0 = -5 volts, and 4095 = +5 volts,

therefore, 3.5 volts = (3.5 - (-5)) / 10 * 4096

= 3481.6

~ 3482

ex: CMD > Aoutput: channel = 3, dac = 3482;

3.2.6.5 - Data file options peculiar to the LDA Environment

FILE : Format = (options), to_file_# = n9 ;

where options can be any combination of the following separated by a comma:

LDA	: generate the current LDA data
Ainput	: produce data read by the ISAAC A/D unit
Aoutput	: produce the ISAAC D/A setting
Traverse	: X, Y, Z position of the traversing unit

3.3 - CADA-CTA Demonstration Program 

Suppose a particular experiment requires 50 sets of data to be read from a multi-channel CTA setup. This requires the RMS and MEAN data from channel 0 and 1 approximately every 2.5 min. The result of these readings must be stored in a text file called "RmsMean0.dat" and "RmsMean1.dat" respectively. In addition, the full status of the RMS and MEAN information must be sent to the printer as the experiment progresses. A FORTRAN program (called ANALYSIS.exe) is then used to analyze each set of data and the output is redirected to the printer.

```
ENABLE: CRMS, CMEAN ;
;
FILE: OPEN   = RmsMean0.dat, TO_FILE_# = 1;
FILE: FORMAT = (TIME, CRMS=DATA, CMEAN=DATA), TO_FILE_# = 1;
;
FILE: OPEN   = RmsMean1.dat, TO_FILE_# = 2;
FILE: FORMAT = (TIME, CRMS=DATA, CMEAN=DATA), TO_FILE_# = 2;
;
FILE: OPEN   = PRN, TO_FILE_# = 9;
FILE: FORMAT = (TIME, CRMS, CMEAN), TO_FILE_# = 9;
;
DO : 1, TIMES = 50;
      CRMS : CHA=0;                      set and read cha. 0
      CMEAN: CHA=0;
      WAIT : DELAY = 150;
      READ : CRMS, CMEAN;
      FILE : STORE, TO_FILE_# = 1;
      FILE : STORE, TO_FILE_# = 9;
      ;
      CRMS : CHA=1;                      set and read cha. 1
      CMEAN: CHA=1;
      WAIT : DELAY = 150;
      READ : CRMS, CMEAN;
      FILE : STORE, TO_FILE_# = 2;
      FILE : STORE, TO_FILE_# = 9;
END: 1;
;
FILE: CLOSE, TO_FILE_# = 1;
FILE: CLOSE, TO_FILE_# = 2;
FILE: CLOSE, TO_FILE_# = 9;
;
DOS := ANALYSIS < RmsMean0.dat > PRN ;
DOS := ANALYSIS < RmsMean1.dat > PRN ;
```

The above program listing will allow the user to solve the given problem. This program can either be entered in the interactive window within the CADA-CTA program or written and stored in a text file (i.e., demo.cmd) before entering the CADA-CTA program. In this case, the user can execute this command within DOS to invoke the computer to perform the acquisition and data analysis automatically as shown in 1 below, or follow 2 to enter the CADA program and execute the command file.

1) C:> CADA-CTA execute : file = demo.cmd ; !

or

2) C:> CADA-CTA

CMD > execute : file = demo.cmd ;

3.4 - CADA-LDA Demonstration Program

In this example, the user requires a three dimensional flow profile over a plane 900 mm in the y axis by 500 mm in the z axis, and each measurement point is 10 mm apart. The LDA counter processor is to acquire a set of readings at approximately 1 minute intervals. Each set of data consists of 10 consecutive readings 0.1 second apart. This also requires the LDA unit to run at mode 1 (acquiring the Doppler frequency and the sample interval time). These readings are stored in a text file called 'LDA.dat'. In addition, the full status of the LDA and traversing unit are required to be printed as the experiment progresses. A plot program (called LdaPlot.exe) is then used to analyze and plot the result to the screen.

```
; NOTE :    text after ';' is considered as a comment and
;          will not be execute.

; -   Enable the LDA unit and configure it to read 10
;     consecutive readings every 0.1 second.

ENABLE : LDA ;
LDA : NSETS = 10, INTERVAL = 0.1 ;

; -   Open and set the format of the data and printer file.
FILE: OPEN   = LDA.dat, TO_FILE_# = 1 ;
FILE: FORMAT = (TIME, LDA, TRAVER), TO_FILE_# = 1 ;
FILE: OPEN   = PRN, TO_FILE_# = 9 ;
FILE: FORMAT = (TIME, LDA, TRAVER), TO_FILE_# = 9 ;

; -   The next statement will cause the computer to pause
;     and wait for a key press.  During this time the user
;     should position the sensor to the measurement origin.
WAIT: KEY ;

; -   The next two statements will enable the computer to
;     communicate with traversing unit and zero the x, y
;     and z counters.
ENABLE : TRAVER ;
TRAVER : = INIT ;

; -   This statement will allow the program to wait till 10
;     pm then start the experiment.
```

```
WAIT : 22:00:00 ;
DO : 1, TIMES = 90 ;                    - 90 measurements in y axis
  ;
  DO : 2, TIMES = 50 ;                  - 50 measurements in z axis
    TRAVER: Z=i+10 ;                    - increment 10 mm in z axis
    WAIT : DELAY = 60 ;                 - delay for 1 min ( 60 sec. )
    READ : LDA ;                        - read data from the LDA unit
    FILE : STORE, TO_FILE_# = 1 ;       - store data to LDA.dat
    FILE : STORE, TO_FILE_# = 9 ;       - send data to printer
    ;
  END : 2 ;
  TRAVER : Y=i+10, Z=0 ;                - move probe to next row
END : 1 ;

FILE ; CLOSE, TO_FILE_# = 1 ;           - close the data file
FILE : CLOSE, TO_FILE_# = 9 ;           - close the printer file

DOS := LdaPlot < LDA.dat ;              - execute the plot program
```

Save the above command code to a text file called LDA_DEMO.cmd. Then issue the following DOS command to perform the acquisition.

1) C:> CADA-LDA execute : file = LDA_DEMO.cmd;

or

2) C:> CADA-LDA

CMD > execute : file = LDA-DEMO.cmd ;

CHAPTER 4 - HARDWARE INTERFACE FOR THE LDA COUNTER PROCESSOR

This chapter describes the PC-LDA interface unit. This unit is mainly intended for data communication between the IBM-PC computer and the Dantec LDA Counter Processor Unit, Dantec part # 55L90a [9]. Figure 4.1 shows the block diagram of the PC, the PC-LDA interface and LDA counter processor.

This chapter is subdivided into three sections. The first section describes the hardware requirement of the Dantec Laser Doppler Anemometer and the design of the PC-LDA interfacing system. The second section discusses the different methods of data acquisition (namely software polling, interrupting and Direct Memory Access) and their differences. Speed considerations are also discussed and examples are given to aid the understanding of the three different types of data acquisition and the setup of the PC-LDA interface hardware (source listings are located in Appendix E and Appendix F contains the PC-LDA circuit diagrams). The last section explains the hardware design and software design initialization requirements to use the PC-LDA interface.

4.1 - Design of the PC-LDA Interface Card

Figure 4.2 shows the back panel of the 55L90A LDA-Counter Processor unit. The two digital input/output ports are located at the left side of the panel designated Digital I/O No. 1 and Digital I/O No.2. Both are 40-pole multi-connectors and are designed for flat cables. The upper terminal in the flat cable is labelled pin No. 1.

Within these two connectors, 49 bits of useful signal are available; one 8 bit and three 12 bit binary output data words (they are labelled F, T, D and P), 1 bit of control output (+DATA READY, "+DR"), and 4 bits of control inputs (-INHIBIT "-INH", +ARM, S0 and S1), while +DR, -INH and ARM are the control lines used to do the handshaking between the PC and the LDA units. The physical pin configuration is shown in Fig. 4.3.

Digital I/O No. 1 consists of two 12 bit binary words, one word of Doppler frequency data, labelled D, and one word of sample interval, labelled T. The sample interval is the time elapsed from one validated velocity value to the next, and it is equal to the time interval between two successive "DATA READY" pulses.

Digital I/O No. 2 consists of one 8 bit binary word of Fringe Number, labelled F, and one 12 bit binary word of Burst Time/Transit Time, labelled P. The T, D, F, and P data are presented simultaneously.

The DATA READY control output line indicates that the D and T data are stable when high ('1'), and the D and T data are being loaded into the output register when low ('0').

There are also four control input lines. One is the -INH (inhibit) signal line which is active low, indicated by the "-" sign in front of

"INH" and it is located in both of the digital connectors. This control line is used to stop the counter by inhibiting the compare pulse when low ('0'). If these lines are not grounded or if they are unused, the counter is automatically enabled.

The second control input signal is designated ARM which is located in Digital I/O port No. 1. As long as ARM is high, the counter is enabled. When the ARM goes low, a single output takes place before the LDA-Counter Processor is disabled. A short positive pulse on the ARM will enable the LDA-Counter Processor for a single measurement.

There are also two timebase select lines (S1 and S0). If they are unused, the timebase for the sample interval measurement will become 100 nano-seconds. This is equivalent to a binary '11' input. If line S0 is grounded, the timebase is 1 milli-second. If both lines are low ('0'), an external timebase is enabled.

4.2 - Method of Data Acquisition

With this PC-LDA interfacing system, a few basic types of interfacing techniques are possible. They are:

1) **Polling**

The software checks the device continuously for its status. This requires the full attention of the microprocessor.

2) **Interrupt**

Run as an interrupt service routine. It will accommodate moderate data acquisition speed, and

3) **Direct Memory Access (DMA)**

The data acquisition process is occurring at maximum speed and is totally transparent to the user's program.

The most common method of interfacing a microcomputer system such as the PC, is through the use of programmable digital input and output registers. With digital output registers, the microprocessor can write data into the register and treat the register as an output port or a memory location. The output of these registers can then be wired to an interface device such as a relay (or the LDA-Counter Processor). Thus, by writing to an output register, it is possible to activate or deactivate a relay. The relay can in turn control, for example, the power to a motor, start or stop an event or simply transfer data to an external device. The digital input registers are very much like the output registers except that they are used to read the status or information attached to their inputs. These input registers can also be treated as a memory location or an input port. For example, for a program required to determine if a switch is opened or closed, the switch can be tied to

the input of a digital input register and its state can be read and determined. Therefore, the data results reflect the state of the signal on the wires. When one is writing to a digital output port, information is transferred to the external device. In general, digital input and output registers allow the microprocessor to send information to and from the outside world. This technique requires the user's software to monitor the external control signal to direct the flow of data. This method requires the processor to spend a considerable amount of time waiting for an event to occur, which implies the processor is idle while it is waiting for the external signal to direct its action. The best data transfer rate which one could expect is approximately 86 kilobytes/sec if it is written in Assembler or 0.2 kilobytes/sec if it is written in Basic.

The hardware interrupt method is very useful and often necessary when interfacing to a microprocessor system. The major advantage of the interrupt method is the ability to get the attention of the microprocessor for service of a function, without requiring the processor to poll an interface constantly for work requests. This leaves the processor free to do other things until it is specifically required by the interface. A good example of using interrupts is for acquiring data from the LDA counter processor if a slow data rate is expected. The computer can do other tasks such as averaging and also receive data at the same time. The maximum data transfer rate is approximately 78 kilobytes/sec if it is written in Assembler, Pascal or C. If the interrupt method is not employed, the microprocessor will always be in a loop waiting for additional data.

In some situations where the LDA-Counter Processor is generating a very high data rate, the software polling and interrupting method are not sufficiently fast enough. To solve this high data rate problem, a special mode of data transfer called Direct Memory Access (DMA) is provided. The DMA mode allows this interface to read and write data to or from memory without using the microprocessor. This function is provided by a DMA controller device. In the IBM-PC design, this device is an Intel 8237-5 DMA controller chip.

maximum DMA data transfer rate is approximately 500 kilobytes/sec if the single byte transfer mode is used. For more details on the DMA controller, refer to Intel Microsystem Components Handbook [16].

Table 4.1 shows the maximum data transfer rate which one can achieve with the different acquisition methods and also gives some idea of CPU overhead. Depending on the setup and/or flow situation, one may be required to select one of the three data acquisition techniques. If the data rate generated by the LDA-Counter Processor is low, and the sole purpose of the program is to read data, then the software polling method of data acquisition is the simplest one to use. If one wants his/her program to have more processing power, it may be a good idea to consider the use of the interrupt method. With this method, the user can set up a main supervisory routine to monitor the entire program, while the interrupting sub-program does the actual data acquisition. An example to this is to write a main program that only does data reduction and plotting, while periodically checking to see if new data are saved by the interrupting routine. If there are new data, the main program can pick them up for processing. In this way, the main program does not have to

wait for data to become available. It can do the handshaking procedure immediately in order to receive the external data.

The most efficient method of data acquisition, however, is Direct-Memory Access (DMA) and not software polling or the interrupt method. With this method, one can achieve the maximum degree of transparency and it also results in the highest rate of data transfer to or from an external device such as the LDA-Counter Processor. In this PC-LDA interface design, the user can have as high as 500 kilobyte of date transfer from the LDA system per second. With the DMA method, the user is required to do some hardware setting. For a detailed procedure of how one may be able to accomplish this, the section on DMA in the software example can be consulted.

Appendix E contains three example programs. One example for each of the data acquisition methods. Those examples programs make use of the information continued in the remainder of this chapter.

4.3 - Description of the PC-LDA Interface Card

4.3.1 - Hardware Description

This system consists of two main units, the signal buffer interface unit and the PC-LDA interface unit. A graphical representation of these two units are shown in Fig. 4.4. The complete circuit diagram and related information are located in Appendix F. The first unit is the PC signal buffer interface which is located at the lower left corner, this interface is used to buffer the signals going to and coming from the IBM-PC system bus, and also increase the strength of the signal so that the signals can be transmitted through a long flat cable.

The second unit, which is located at the right side of the figure, is a block diagram of the actual circuit designed to communicate between the IBM-PC computer and the Dantec LDA Counter Processor. One can divide it into a number of smaller units, namely, Address Decoding Unit, System Command Registers, System Function Ports, Data Ports and Counter/Control Logic which are described below.

4.3.1.1 - Address Decoding unit.

This unit is used to decode the PC address line A0 to A4 and generate the appropriate device select pulses. With these pulses, one can select a number of sub units, and they are:

1) unit 2- System Command Registers,

2) unit 3- System Function Ports and

3) unit 4- the Data Ports.

4.3.1.2 - System Command Registers

These registers are used to configure the operation of this interfacing system and also allow the user to monitor some of the important information within the system. A more detailed description of these registers will be discussed later.

4.3.1.3 - System Function Ports

When the appropriate address and control signal are selected, four different types of functions can be generated from this unit. They are:

1) Software generation of a reset signal: This is used to reset the Intel 8255A-5 chips (generally referred as a programmable peripheral interface chip) to a known state.
2) Generate an ARM Pulse used by the LDA-Counter Processor to read the next set of Doppler signals.
3) Generate a Clear INHibit Pulse: This pulse is used to clear the handshaking signal required by the LDA-Counter Processor to free its output data.
4) Sequential Read/Write Port: With this port, the user can read or write a maximum of 48 bits (or 6 bytes) of data sequentially, 8 bits (or 1 byte) at a time. However, for this port to function correctly, the user is required to do some initialization before hand. A detailed outline of what must be done will be discussed in the PC-LDA interface initialization requirement section.

4.3.1.4 - Data Ports (1 to 6)

These six data ports are located within two Intel 8255A-5 IC chips. The direction of these six data I/O ports are set by the appropriate Intel 8255A-5 command register. The user can address each of these data ports separately via their individual port address, or sequentially read/write via a special port discussed previously (Sequential Read/Write Port).

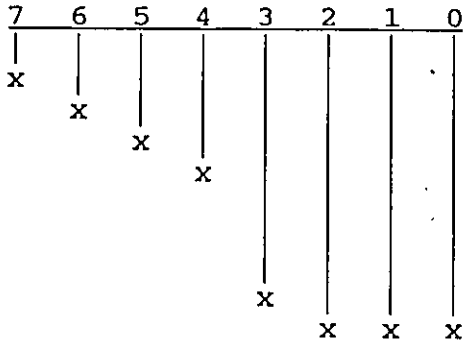
4.3.1.5 - Counter and Control Logic Unit

This unit is used to generate internal system signals and controls the data transfer between the IBM-PC computer and the LDA unit.

4.3.2 - Bit Configuration of System Command / Function / Data Registers

These I/O registers are used to control and monitor the LDA counter processing unit. Table 4.2 contains the complete I/O locations that the PC-LDA rely on. The following is the exact bit configuration of the system command registers. The first register (Base +0, which is an output register), is the most important one. This register contains five pieces of information. It must be initialized before any actual data communication can be performed. The following is the bit wise configuration of this register.

1) System Command Register #1 (Base +0)



DMA enable or disable flag
Interrupt enable or disable flag
Arm pulse enable or disable flag
INHibit enable or disable flag
0 : disable.
1 : enable.

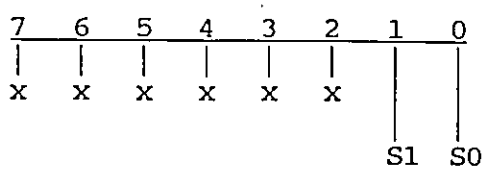
not used
contains the counter information

Bits 0 to 2 are required only if sequential data read/write is used. If sequential read/write is not used, these 3 bits have no effect. The following lists the port access sequence.

bit: 2 1 0
0 0 0 : selected data port 1 to 6.
0 0 1 : selected data port 2 to 6.
0 1 0 : selected data port 3 to 6.
0 1 1 : illegal.
1 0 0 : selected data port 4 to 6.
1 0 1 : selected data port 5 to 6.
1 1 0 : selected data port 6 to 6.
1 1 1 : illegal.

The next register is the system command register number 2, (this has the I/O port address of Base + 1). It is an output register. It is used to send the Timebase information to the LDA-Counter Processor, and its bit configuration is as follows:

2) System Command Register #2 (Base +1)



unused

Time base setting S1 then S0
0 0 : external time base used.
0 1 : not used.
1 0 : 1 milli-second.
1 1 : 100 nano-second.

The third system command register (Base + 2) is configured for input. With this register, the user can determine the state of the DATA READY line from the LDA-Counter Processor, and it is wired directly to bit 7. This register is specially designed for software polling application since the software requires some information to inform the program that the external data are stable for reading. The pin configuration of this register is as follow:

3) System Command Register #3 (Base +2)

bit:	7	6	5	4	3	2	1	0
DATA READY signal from LDA unit 0 : Data not ready. 1 : Data ready for reading	x							
unused		x	x	x	x	x	x	x

The last of the system command registers is number 4. This register is used to configure the 8255-0 IC chip. Detailed information reading this register can be found in the Intel Microsystem Components Handbook [16] under operational description of Intel 8255A-5.

For the system function ports, one can use either a read or write I/O port command to activate it. However, the six data ports are slightly different. The read I/O port command reads a byte of data from that location and a write I/O port command transfers a byte of information to it. The following are the bit configurations of these six data ports.

1) Data port #1 (Base +8)

bit:	7	6	5	4	3	2	1	0
Burst time/Transit time mantissa	Pm7	Pm6	Pm5	Pm4	Pm3	Pm2	Pm1	Pm0

2) **Data port #2 (Base+9)**

bit:	7	6	5	4	3	2	1	0
Burst time/Transit time exponent	Px3	Px2	Px1	Px0				
unused					x	x	x	x

3) **Data port #3 (Base +10)**

bit:	7	6	5	4	3	2	1	0
Fringe number	F7 MSB	F6	F5	F4	F3	F2	F1	F0 LSB

4) **Data port #4 (Base +12)**

bit:	7	6	5	4	3	2	1	0
Sample interval mantissa	Tm7	Tm6	Tm5	Tm4	Tm3	Tm2	Tm1	Tm0

5) **Data port #5 (Base +13)**

bit:	7	6	5	4	3	2	1	0
Sample interval exponent	Tx3	Tx2	Tx1	Tx0				
Doppler frequency exponent					Dx3	Dx2	Dx1	Dx0

6) **Data port #6 (Base +14)**

bit:	7	6	5	4	3	2	1	0
Doppler frequency mantissa	Dm7	Dm6	Dm5	Dm4	Dm3	Dm2	Dm1	Dm0

The 8 bit binary word of fringe number "Nf" is arranged as a positive integer, and it is represented as follows:

Nf= F7 F6 F5 F4 F3 F2 F1 F0

The 12 bit binary words are arranged as a read number; 8 bit mantissa and 4 bit exponent. There is an example of how the doppler frequency "fD" is represented in base 10 (decimal) notation. (T and P are calculated in similar way.)

D = Dm7 Dm6 Dm5 Dm4 Dm3 Dm2 Dm1 Dm0 * 2 exp(Dx3Dx2Dx1Dx0)

The actual Doppler frequency can be calculated from the following formula

$fD = (10^9 / (255*2^{18})) * D$ (Hertz)

$fD = 14.96 * D$

$fD = 0.5348 / (\text{Burst time})$

4.3.3 - PC-LDA Interface Initialization Requirement

In order for this interfacing system to function correctly, a number of system initializations are required. The first initialization step is to reset the three Intel 8255A-5 chips. The sole purpose of this is to put them into a known state.

Secondly, with the initialization of the various 8255A-5 command registers, one can program the direction of data transfer of the individual data port. For example, one can set the six data ports to all input or all output or partly input and partly output (when interfaced to the LDA unit all of the ports must be set as input). For more details, please refer to the section of operational description of Intel 8255A-5 in the Intel Microsystem Components Handbook.

The next step is to clear the -INH (inhibit) line. This is required to ensure that the PC-LDA interface unit does not inhibit the LDA unit.

The loading and setting of the interface system command registers are next. In this stage, the user is required to load and set the system counter if sequential read/write is used. If not, loading and setting of counter can be skipped.

Lastly, the user must select the mode of data transfer. This is simply done by setting the various bits in the command register number 1. The various modes the user can choose from are software dedicated polling, interrupting or DMA (Direct Memory Access).

This completes the system initialization of the interfacing system. The next section will give a step by step layout of the complete initialization required.

4.3.3.1 - Step by Step Initialization Procedure

The I/O port locations contain the physical port address of the various functions. One can execute them from the PC-LDA interface card, and they are:

```
1) Reset        = Base+ 4;          { used to reset the 8255s      }
2) PPI_0        = base + offset[0]; { starting location of 8255-0  }
3) PPI_1        = base + offset[1]; { starting location of 8255-1  }
4) PPI_2        = base + offset[2]; { starting location of 8255-2  }
5) Clear_INH    = base + 6;         { clear INHibit and DMA        }
6) Sys_Com_Reg  = PPI_0;            { port A of 8255-0             }
```

Another command which will be used throughout is the "port[ ]" command, this command is supplied by Borland's Turbo Pascal Language [19] to access the hardware. With this command, one can read or write a byte of binary information from or to any standard Input/Output port in the IBM-PC system.

An example which illustrates the uses of this command for writing or reading the system command register is

```
temp_storage := port[Sys_com_reg];
```

Now, there is an example procedure used to initialize the system for data acquisition, "procedure init_lda_interface (count);". The parameter count is equal to the number of data ports one wants to read/write through the sequential read/write port. The procedure can be found in Appendix E, in the source listing of PC-LDA initialization routine.

CHAPTER 5 - Discussion and Recommendations

CADA-CTA and CADA-LDA are data acquisition environments which are used to computerize the data recording, position the sensor and control the instruments. They use the same procedure as the manual method to acquire the experimental results. The environments are very generic, flexible and affordable. It will give the experimenter a capacity for fast automated and remote data recording and control of the CTA and LDA instruments. The data recording is done internally, therefore, this eliminates human round off, and manual recording errors. Since the computer is actually doing the data recording, the number of data points is not a concern. The experimenter can spend more time to solve other problems.

The two software environments are very easy to learn and operate. The user is presented with a very user friendly interactive windowing environment. A help facility is always on line. If the user ever encounters a problem, the solution is a key stroke away.

This PC-LDA interface card is designed to communicate between the IBM personal computer and Dantec's LDA counter processor unit. Since the IBM personal computer is widely used, the CADA-CTA and CADA-LDA can be used in a number of laboratories. In addition, the PC-LDA interface card can be used independently.

The following improvements to CADA-CTA and CADA-LDA are suggested. The interactive command language can be enhanced by adding more logic control commands such as IF and WHILE statements. It will also make the

language more complete if variables and arithmetic operations are included.

The current software has been developed using TURBO PASCAL version 3.0 which has a program size restriction. For this reason, the CTA and LDA environments have been developed separately. By porting to TURBO PASCAL version 4.0 or C language, which does not have such a restriction, the two environments can be combined. As an alternative, the device drivers can be made as separate load modules, with only the load module of interest used at any one time.

REFERENCES

1. 56C01/17 Constant Temperature Anemometer, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

2. 56N10 RMS Value Unit, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

3. 56N11 Mean Value Unit, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

4. 56N20 Signal Conditioner, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

5. 56N12 Correlator Unit, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

6. 56N22 Mean Value Unit, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

7. 56N25 RMS Value Unit, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

8. 56N30 Signal Analysis Module, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

9. 55L90a Laser Doppler Anemometer Counter Processor, Dantec Documentation Department, Scientific Research Equipment Division, Denmark, (1984).

10. David C. Willen & Jeffrey I. Krantz, 8088 Assembler Language Programming, The IBM PC, Howard W. Sams & Co. (1983).

11. James W. Coffron, The IBM-PC Connection, Sybex Inc., Berkerly, (1984).

12. Murray Sargent III & Richard L. Shoemaker, The IBM Personal Computer From the Inside Out, Addison-Wesley Publishing Company, Canada, (1984).

13. IBM-PC Technical Reference Manual, International Business Machines Corporation, Florida, (1984).

14. IBM-PC Technical Reference Manual, International Business Machines Corporation, Florida, (1984).

15. IBM-AT Technical Reference Manual, International Business Machines Corporation, Florida, (1984).

16. Microsystem Components Handbook Volume I and II, Intel Corporation,

Santa Clara, (1984).

17. Schottky TTL Data Book, Motorola Semiconductor Products Inc., Phoenix, (1983).

18. David C. Willen & Jeffrey I. Krantz, 8088 Assembler Language Programming The IBM PC, Howard W. Sams & Co., Inc., (1983).

19. Turbo Pascal Version 3.0 Reference Manual, Borland International Inc., Scotts Valley, (1985).

20. ISAAC Labsoft Software & Hardware Documentation, Cyborg Corporation, 55 Chapel Street, Newton, MA., 02158. (1985)

21. National Instrument IEEE-488 Interface, National Instrument, 12109 Technology Boulevard, Austin, Texas, 78727-6204, (519)250-9119. (1986)

FIGURES

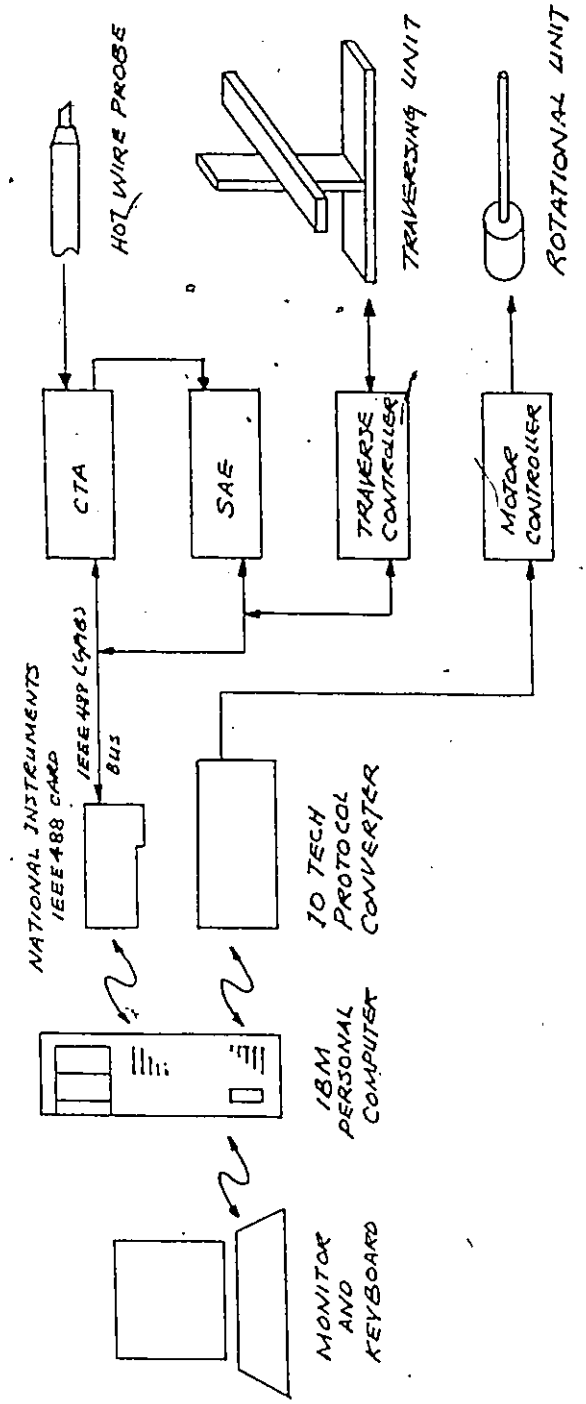


Fig. 1.1 - Configuration of the CADA-CTA Environment

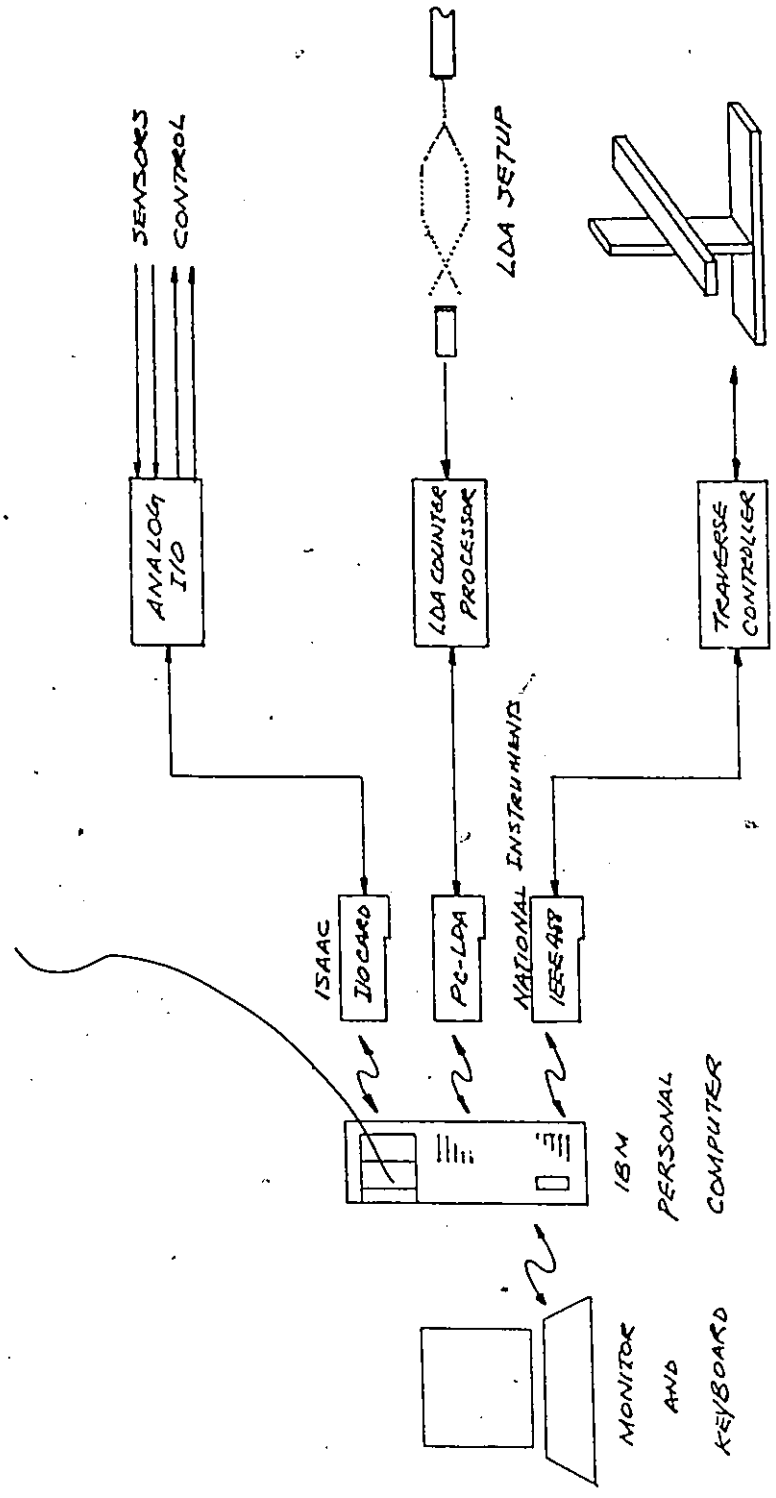


Fig. 1.2 - Configuration of the CADA-LDA Environment

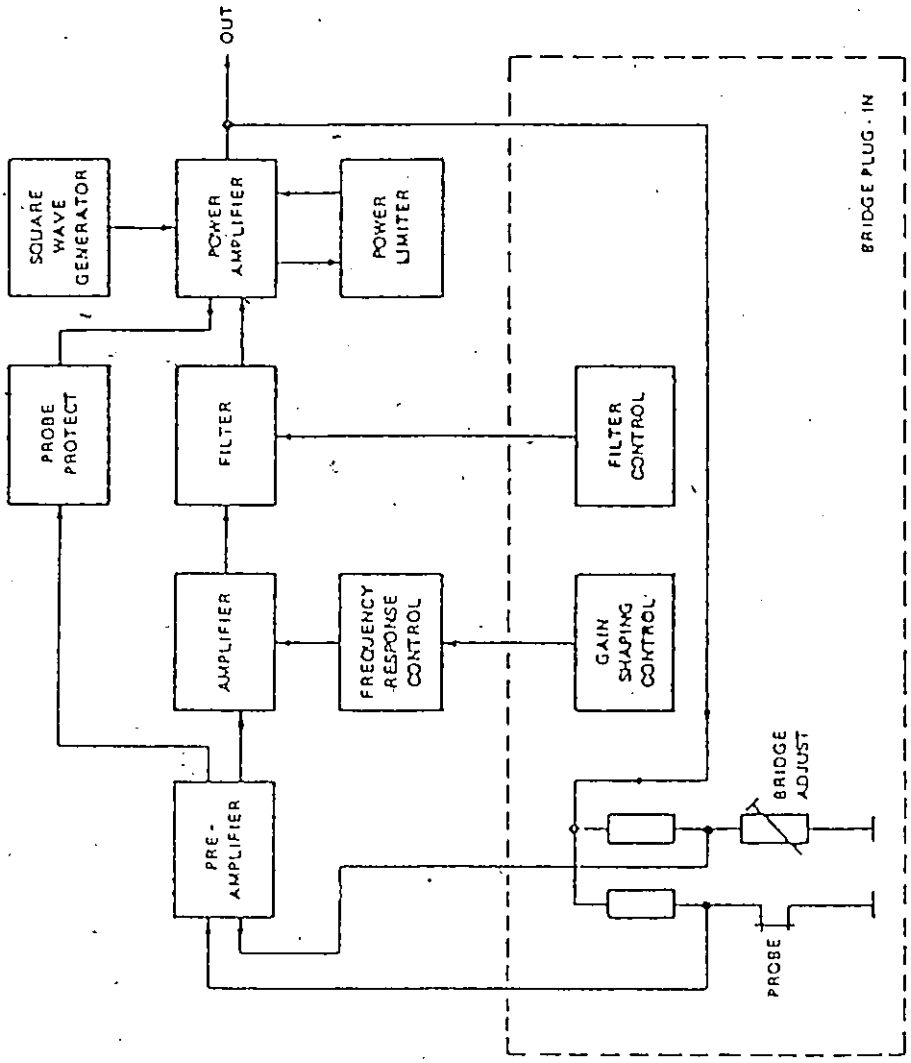


Fig. 2.1 - Block Diagram of Constant Temperature Anemometer (56C01/C17) [1]

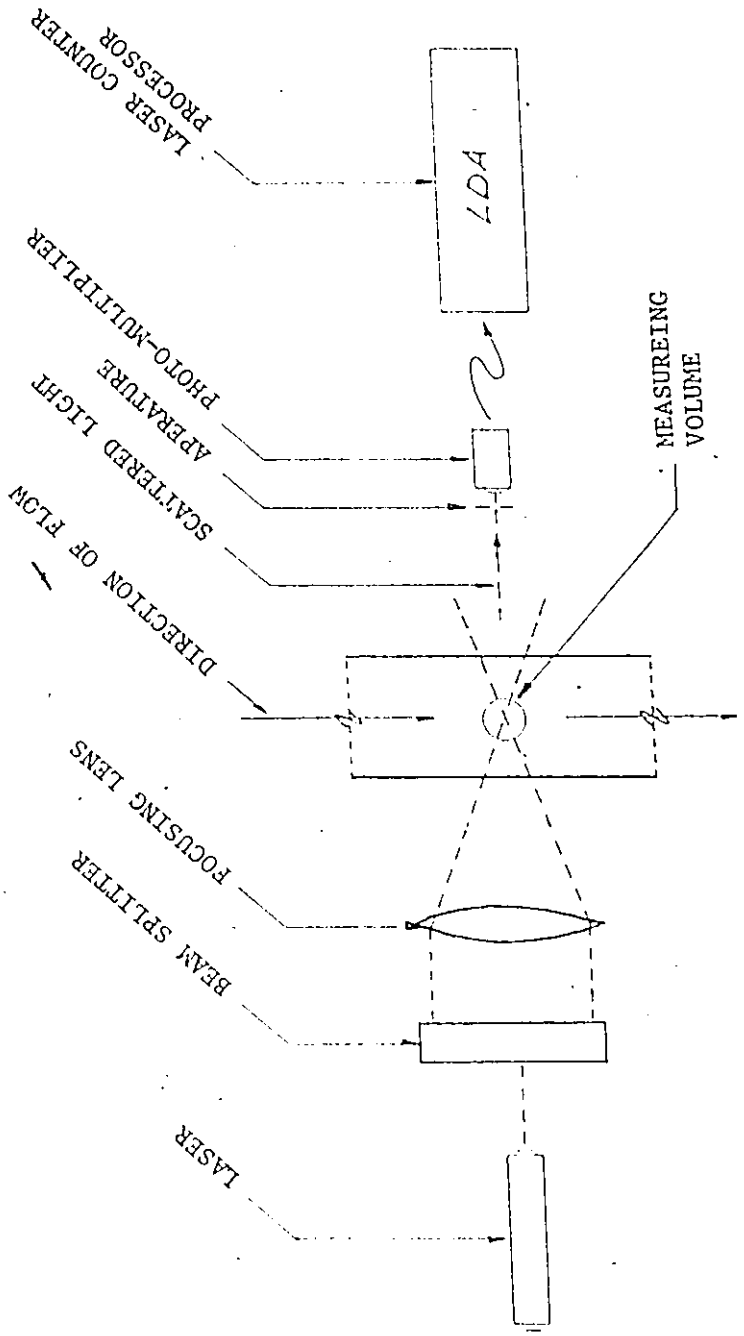


Fig. 2.2 - Schematic of Forward Scatter Laser Anemometer Flow Measurement System

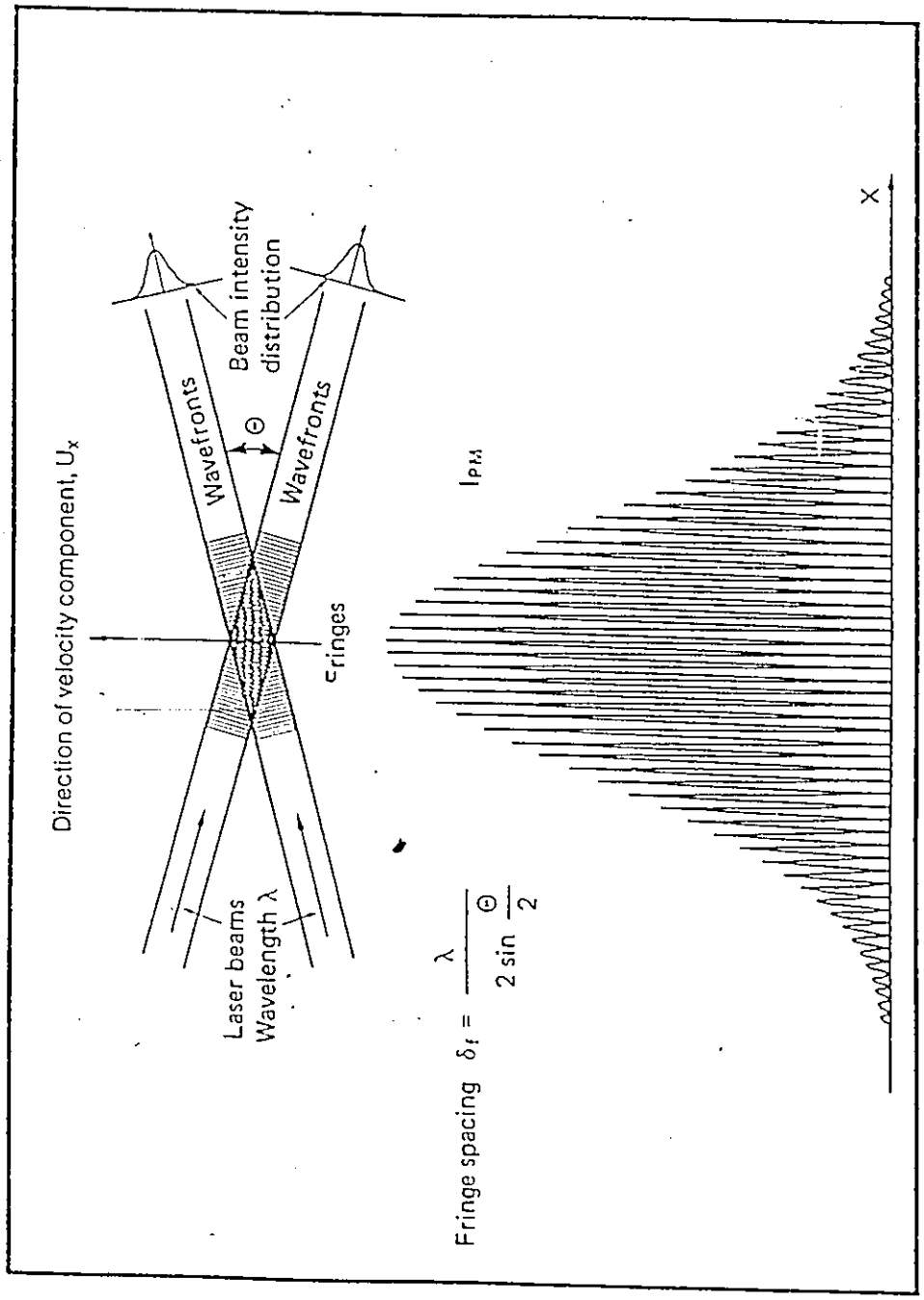
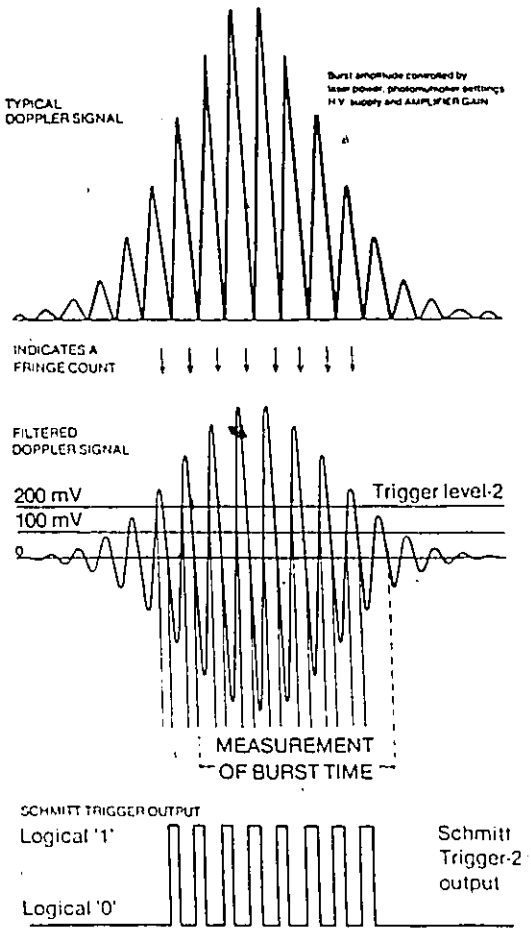


Fig. 2.3 - Two intersecting laser beams forming an interference or fringe pattern at the intersection [9]



Doppler signal from a signal scattering particle and the output from the Schmitt Triggers.

Fig. 2.4 - Principle of the LDA Counter [9]

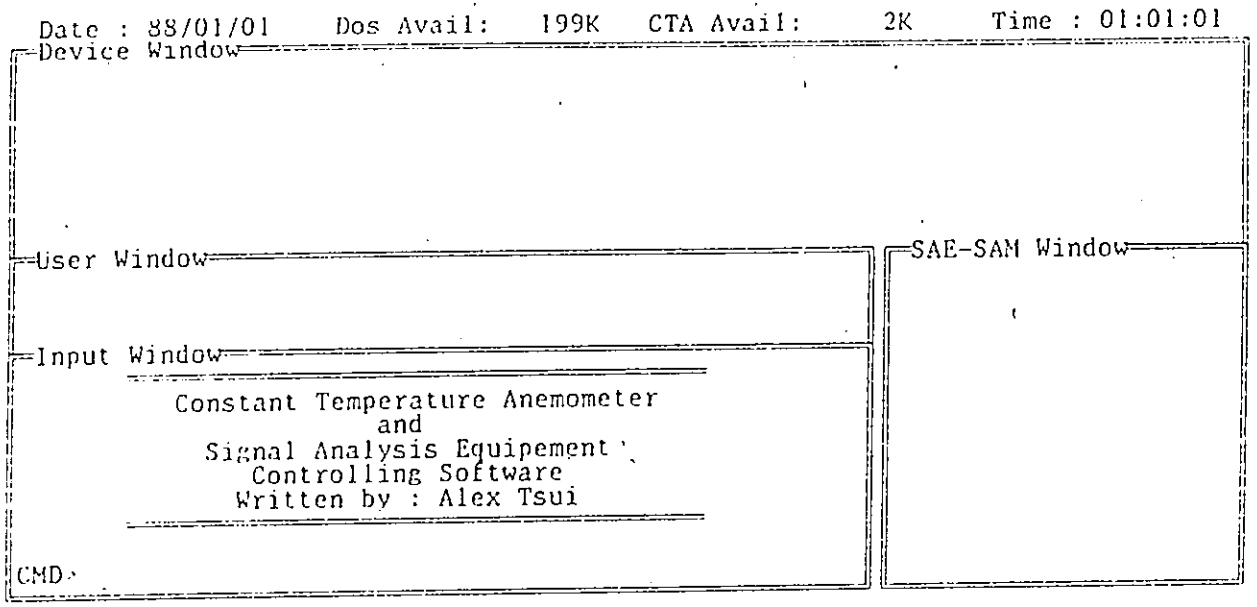


Fig. 3.1 - Start Up Display of CADA Environment

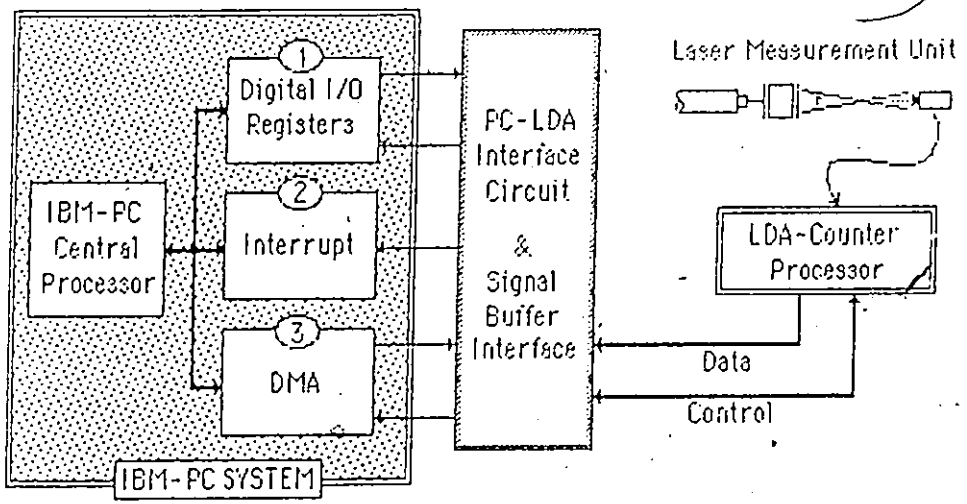


Fig. 4.1 - Block Diagram of the PC-LDA Interface

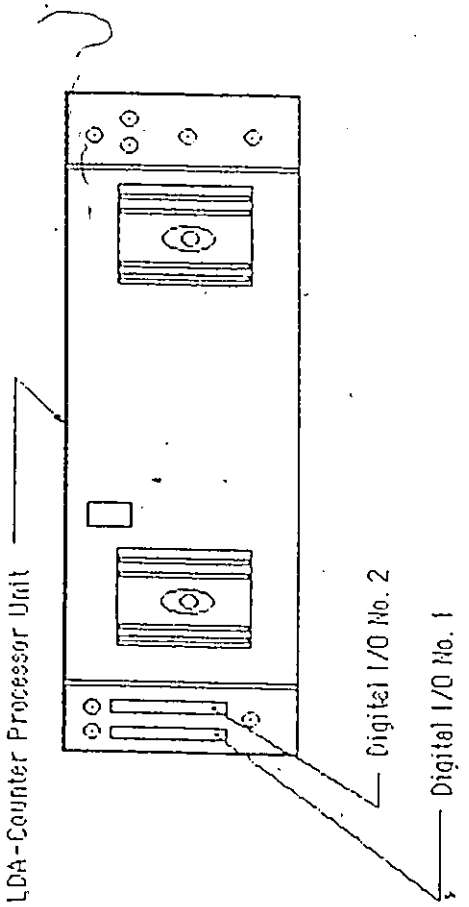
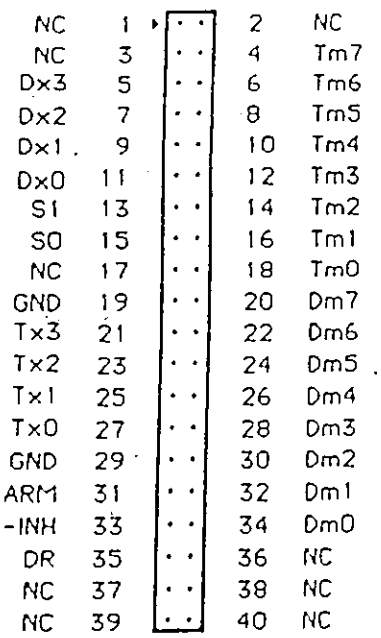
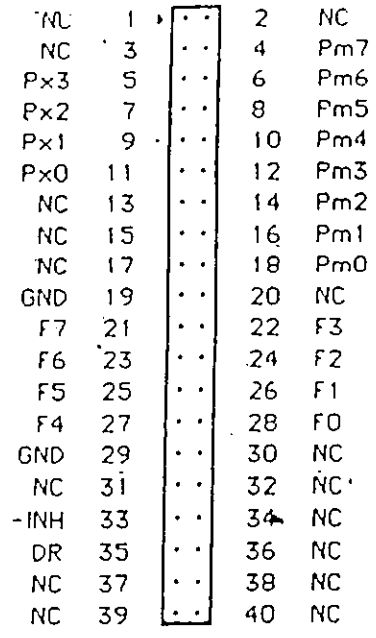


Fig. 4.2 - LDA Counter Processor's Back Panel



Digital I/O No. 1



Digital I/O No. 2

Fig. 4.3 - LDA's Digital I/O Port pins layout

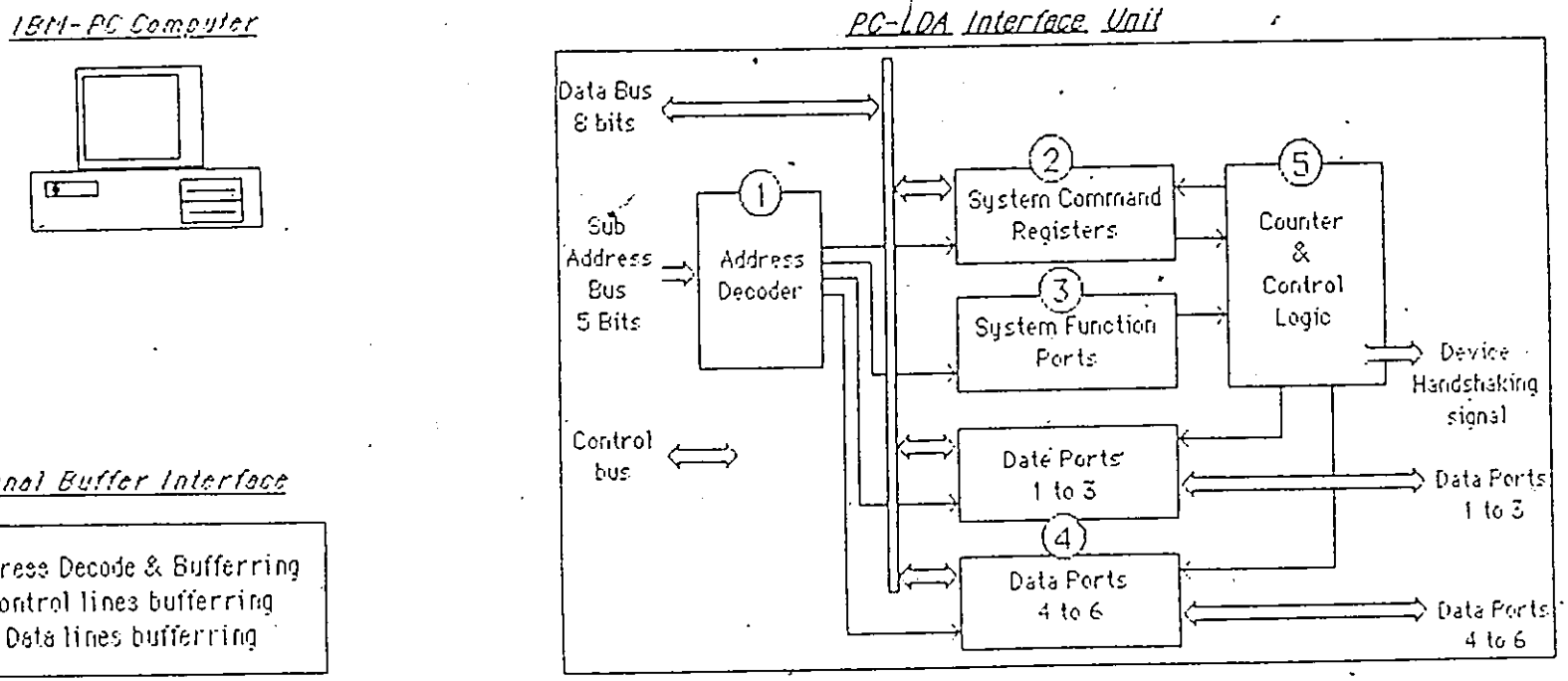


Fig. 4.4 - System Block Diagram

TABLES

TABLE 4.1 - Speed and Overhead

Method	Maximum Transfer Rate	CPU Overhead
Programmed I/O	86 kilobytes/sec.	Full time
Interrupts	78 kilobytes/sec.	Part time
DMA (single byte transfer)	500 kilobytes/sec.	Little

TABLE 4.2 - System I/O Location

Standard I/O address	Function Input (read)	Function Output (write)
System command registers		
1) Base + 0	input from Port A0	output to Port A0 **
2) Base + 1	input from Port B0	output to Port B0 **
3) Base + 2	input from Port C0 **	output to Port C0
4) Base + 3	not available	8255-0 Command Register
System function ports		
1) Base + 4	Software Reset	
2) Base + 5	ARM pulse	
3) Base + 6	Clear Inhibit Pulse	
4) Base + 7	Sequential Read/Write Port	
Data port 1 ,2 and 3		
1) Base + 8	input from Port A1**	output to Port A1
2) Base + 9	input from Port B1**	output to Port B1
3) Base + 10	input from Port C1**	output to Port C1
Base + 11		8255-1 Com Reg
Data port 4, 5 and 6		
4) Base + 12	input from Port A2**	output to Port A2
5) Base + 13	input from Port B2**	output to Port B2
6) Base + 14	input from Port C2**	output to Port C2
Base + 15		8255-2 Com Reg

NOTE: ** : Used for PC-LDA interface configuration.
Base : Is equal to hex 8300 in this system.

APPENDIX A - COMMAND DESCRIPTION FOR CADA-CTA

Interactive CTA-SAE-TRAVER controlling system

Written by : ALEX TSUI
Last Update : 18:09:42 2/4/1987

University Of Windsor
Department of Mechanical Engineering

Command Syntax

SYNTAX: CTASAE <CTASAE command line>

The syntax used to invoke this program in the DOS environment is shown above.

An optional command line may be included during the invokation of this program. this optional command line will be treated as the first line to be interpreted by this software.

If this option is not included. This software will wait for the first input after it is initalized.

example: To invoke CTASAE software and let the software prompt for input command.

type >> CTASAE

The follow syntax can be used to invoke the CTASAE program and execute a command file to do automatic data acquisition.
(The user must write the command file "acquire.cmd" with an editor before typing in the following line)

type >> CTASAE execute:file=acquire.cmd;

Definition of this help file

```
Primary_Cmd : Sub_Cmd = Value [, Sub_Cmd = Value] ;
   ***      *   ***   *  ***   *   ***   *  ***   *
            ^        ^      ^        ^        ^
```

*** -> Primary or Secondary/Option command.
* -> Separators.

NOTE: Don't forget the SEPERATORS it is essential !!

All Primary command must end with COLON ----------> ' : '.
Secondary command seperator is a COMMA -----------> ' , '.
All command line must terminate with SEMI-COLON --> ' ; '.
All string after ';' will be considered as comment.

n2= Integer value between the range of 0 and 2 (ie 0,1, .. 2)
n7= Integer value between the range of 0 and 7 (ie 0,1, .. 7)
r = Real number (ie 1.128)
Ir= Incremential value (ie +0.25 or -1.115)

Note: 1) Tokens within a set of square barcket are used to indicated that none or one of the specified option can be selected.

ie: If [token1 or token2 or token3] are given, none or one can one can be chosen from that list.

2) Only the FIRST THREE characters of the token or command are important.

3) Blanks is neither a token seperator nor of any important except when a DOS command is made.

Note: <ESC> can be used to exit an executing (command or command file).

DEVICE control commands

```
[Primary] [Secondary Token]
ENABLE:   CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, TRAVER, PROBE, ALL ;
DISABLE:  CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, TRAVER, PROBE, ALL ;
READ:     CRMS, CMEAN, COND, SRMS, SMEAN, SAM, SCORR, TRAVER, PROBE, ALL ;
```

example: To select the MEAN and RMS units from the CTA expansion bus and the SAM unit for the SAE bus, the follow command must be given.

```
CMD > enable: crms, cmena, sam;
```

The follow command will read all the device(s) which one have enabled.

```
CMD > read: all;
```

or to read just one of the enabled device.

```
CMD > read: crms;
```

If one wishes to deselect a device, the DISABLE command should be used.

```
CMD > disable: crms;
```

CTA DEVICE programming command

```
[Primary] [Secondary Token]
CRMS:     CHA        = n7,                    - 1,   2, ...   6
          TC         = r,                     - 0.1, 1, 10, 100 sec.
          DEVICE_NUM = n15 ;                  - 0,   1, ...  15

CMEAN:    CHA        = n7,                    - 1,   2, ...  14
                                                15 =  0.0  volt ref.
                                                16 = 10.0 volts ref.
          TC         = r,                     - 0.1, 1, 10, 100 sec.
          DEVICE_NUM = n15 ;                  - 0,   1, ...  15

COND:     CHA        = n2,                    - 0,   1, ...
          GAIN       = r,                     - 1.0      ... 500 times
          LP         = r,                     - 0.1      ... 300 kHz.
          HP         = r,                     - 0.1      ... 300 Hz.
          DEVICE_NUM = n15,                   - 0,   1, ...  15
          DISPLAY = [ON or OFF] ;             - ON or OFF
```

Note 1 - The default setting aof the DEVICE NUMBER is 0 .
2 - The signal conditioner has an extra sub-command called 'DISPLAY' with this option one can enable or disable the displaying of HP, LP and GAIN setting to the selected channel after a READ or COND command is executed. (default setting is OFF)

example: To program the CTA RMS unit to take readings from channel 1 with a time constant of 10.0 sec. One should type in:

```
CMD > crms: cha=1, tc=10.0;
```

If one wishes to reprogram a particulnr deviec. One should type in a command similar to the one above with the new setting. for example, if one wants to change the time constants from 10.0 second to 1.0 second after the above commands are entered.

```
CMD > crms: tc=1.0;
```

will be sufficient.

example: To select and set channel 0 of the Signal Conditioner to respond to a high pass level of 100, low pass level of 300 and the gain of 100. one must do the following:

```
CMD > enable: cond;
CMD > cond:   cha=0, hp=100, lp=300, gain=100, display=ON;
CMD > read:   cond;
```

SAE DEVICE programming command

```
[Primary] [Secondary Token]
SRMS:     CHA          = n7,                    - 0,   1, ...   7
          CHB          = n7,                    - 0,   1, ...   7
          TC           = r,                     - 0.1, 1, 10, 100 sec.
          DEVICE_NUM   = n15 ;                  - 0,   1, ...  15

SMEAN:    CHA          = n7,                    - 0,   1, ...   7
          TC           = r,                     - 1.049  ... 1049 sec
          DEVICE_NUM   = n15 ;                  - 0,   1, ...  15

SAM:      TRIGER       = n3,                    - 0,   1,  ...   7
          PROCESS      = n31,                   - 0,   1,  ...  31
          ACCTIM       = r,                     - 0.00025 ... 687194 sec.
          TEST         = [0 or 1],              - for H/W testing
          DEVICE_NUM   = n15,                   - 0,   1, ...  15

SCORR:    CHX          = n7,                    - 0,   1, ...   7
          CHY          = n7,                    - 0,   1, ...   7
          IDELAY       = r,                     - Initial delay in sec.
          SDELAY       = r,                     - Step delay in sec.
          TFACTOR      = n                      - Time factor
          DEVICE_NUM   = n15 ;                  - 0,   1, ...  15
```

example: The SRMS and SMEAN commands are very similar to the CTA device commands. The SAM and SCORR commands stands for SAM unit and the CORRELATOR unit connected to the Signal Analysis Equipement bus.

TRAVER DEVICE programming command

```
[Primary] [Secondary Token]
TRAVER:   =INIT ;                               - init. posn. ctr. to 0.0
TRAVER:   X = [r, Ir], Y=[], Z=[], R=[],        - Ir => increment by r mm.
          DEVICE_NUM = n15,                     - device num. for the X axis.
          DELAY= [r or WAIT or ALL_FINISH] ;    - delay before the next traverse
                                                  command will be send.
```

example: The TRAVER: command is designed to initialize and control the positioning of the axes which connect to the TRAVERSING unit.

The INIT subcommand is used to zero the position counters. In order to manually position the traver axes position by adjusting the traver remote control unit to a known datum position in there particular experimental setup, then issue the following:

```
CMD > traver: =init;
```

The X axis is addressed to the device specified by the DEVICE_NUM parameter, where axis Y is one higher then DEVICE_NUM, and Z is two higher then DEVICE_NUM. The default setting for the DEVICE_NUM for all of the CTA and SAE unit are '0'.

To program the traver axes to location X = 10.5mm, Y = 2.125mm and Z = 1.0mm from a datum position and wait until all three axes movement to stablize before executing the next command, the follow should be enter.

```
CMD > traver: x=10.5, y=2.125, z=1, delay=all_finish;
```

if the DELAY=WAIT option is used, the software will only wait for the last specified axis (namely the Z axis).

To increment the traver from the current X location to a new position 100mm in the negative direction, one should issue an icremential offset as the new location.

```
CMD > traver: x=i-100.0, delay=5.5;
```

The delay option specified in the above commahd has signified a 5.5 second delay before interpreting the next input command.

If more then one traver unit is connected to the system, one can issual the following command to select the second traverling unit. Assume the device num (DCBA switch setting) for the first traverling unit is 0, the second traverling unit are set to 8, and both of the units are initialize correctly.

```
CMD > traver: device_num = 8; select dev 8       (2nd Traver)
CMD > read:   traver;         update software posn. counters
CMD > traver: x=100, ..... ;  program second traver unit.
..
..
CMD > traver: device_num = 0; deselect dev 8 and select 0
CMD > read:   traver;         update software posn. counters.
```

PROBE DEVICE programming command

> not implemented in this system <

[Primary] [Secondary Token]
PROBE: =INIT ;
PROBE: ANGLE = [r or Ir] ;

notes: This command is used to rotate the hot-wire probe mounted on a special stepper motor mechanism.

The INIT subcommand used to zero the angle counter.

The ANGLE subcommand can be used to rotate the probe to an absolute or and relative angular position.
(the angular resolution of rotation is 7.5 deg.)

To position the probe to 15 degree from the datum position, the following command are needed.

```
CMD > probe: angle=15;
```

If later, one wanted to increment the current position to 20 deg. from the datum, the following can be used.

```
CMD > probe: angle=20;
or
CMD > probe: angle=I5;
```

DOS LEVEL COMMUNICATION COMMANDS

[Primary] [Secondary Token]
DOS [-p] ;
- Open a subprocess to DOS, type EXIT to return to CTASAE program.
- If the '-p' option is given, CTASAE will bypass the question 'press RETRUN to continue....'

ex : CMD > DOS;

```
C> setpath              : execute a bat program
.. .. .. ..
C> dir/w                : display directory
.. .. .. ..
C> lotus                : run a DOS program
.. .. .. ..
C> exit                 : return to CTASAE program
```

press RETURN to continue ....

DOS [-p] := <d:\path\filename [param.]> ;
- Open a DOS process and execute the given DOS command.
- The DOS command can be a DOS SHELL, f.COM, f.EXE or f.BAT .
- If the '-p' option is given, CTASAE will bypass the question 'press RETRUN to continue....'

ex : CMD > DOS -p := lotus;

After the above line is entered, CTASAE will open a subprocess to execute lotus. At this point, one can edit a work file, etc. But, upon exiting the lotus program, the user will find that he/she is back in the CTASAE parogram.

SYSTEM and MISC. commands

```
HELP ;    - will display this file.
          - one can control the movement of the window with the cursor
            keys (namely : HOME, END, Up, Down, Left, Right, PgUp and PgDn)
          - press F10 to exit the help window.

QUIT ;    - This will close all opened files and return to DOS.

WINDOW ;  - Allow the user to use CURSOR KEY to change window setup.
              F5 - Scroll screen.
              F6 - Size   screen.
              F7 - Move   screen.
              F8 - Change Window  ( this is done in a circular manner )
              F9 - Save,  Recall  or  Delete  window configuration file.
              F10- EXIT   window  setup  menu.
```

LOOPING ans WAIT commands

```
[Primary] [Label] [Secondary Token]
DO:       n9,     TIMES=n255/;              - The label of each DO command
  ..                                          must match with an END
END:      n9 ;                                command of the same label.

WAIT:     KEY,                              - wait for key pressed
          TIME  = hh:mm:ss,                 - wait until hh:mm:ss
          DELAY = r ;                       - delay for r second

example:  To take 100 reading from the sAE-RMS u.it starting a  6:30 pm
          from XY^ = (0, 0, 0) i- every 1]m in the X direc4ion with a 100
          second delay for the RMS reading to be stablized.

              CMD > wait:   time=18:30:00;
              CMD > traver: x=0, y=0, z=0, delay=all_finish;
              CMD > do: 1, times=100;
              CMD >     wait: delay=100.0;
              CMD >     read: srms;
              CMD > end: 1;
```

CMD file EXECUTION commands

```
[Primary] [Secondary Token]
EXECUTE:  FILE=<drive:path\filename.ext> ;  - exec. a sequence of
                                              programmed command.

example:  One can write his own controlling command file with his
          choice of editor and execute them within the CTASAE program. For
          example, if the command of the last example (exclude 'CMD >')
          are stored in a file called READ-RMS.cmd. One can invoke it by
          using the EXECUTE command in the following way.

              CMD > execute: file= read-rms.cmd;

          If this cmd file is located in a different directory, then
          one should include a path name.

              CMD > execute: file= \SAE-dir\read-rms.cmd;
```

FILE handling command

```
[Primary] [Secondary Token]
FILE:     [   OPEN   = <filename.ext>          - open a data file
           or CLOSE                            - close a data file
           or FORMAT =( .. )                   - define write format
           or STORE ],                         - store data to file
          TO_FILE_#  = n9 ;                    - to file number n9


          where ( .. ) = ( TIME, DATE,         - generate time/date
                           SP=n,               - gernerate n space(s)
                           [RMS=DATA  or RMS],  - data or data w/status
                           [MEAN=DATA or MEAN], - data or data w/status
                           COND,               - signal conditioner
                           TRAVER,             - x,y,z position in mm.
                           PROBE,              - angle in degree.
                           SAM )                     - all 8 channel will be
displayed
                                                 (this  unit  must  be  specify
last)
```

example: A particular experiment requires 50 set of data to be read from a multi-channel CTA setup. This required the RMS and MEAN data from channel 0 and 1 in approximate every 2.5 min. The result of these reading are to be stored to a text file called 'RmsMean0.dat' and 'RmsMean1.dat' respectively. In addition, the full status of RMS and MEAN information are to be printed to the printer as the experiment progresses. Then a FORTRAN program (called ANALYSIS.exe) is used to analysis each set of data and print the result out to a printer.

```
CMD > file: open   = RmsMean0.dat, to_file_# = 1;
CMD > file: format = (time, Crms=data, Cmean=data), to file = 1;
CMD > ;
CMD > file: open   = RmsMean0.dat, to_file_# = 2;
CMD > file: format = (time, Crms=data, Cmean=data), to file = 1;
CMD > ;
CMD > file: open   = prn, to_file_# = 9;
CMD > file: format = (time, Crms, Cmean), to file = 9;
CMD > ;
CMD > do : 1, times = 50;
CMD >     Crms : cha=0;                    set and read cha. 0
CMD >     Cmean: cha=0;
CMD >     wait : delay = 150;
CMD >     read : Crms, Cmean;
CMD >     file : store, to_file_# = 1;
CMD >     file : store, to_file_# = 9;
CMD >     ;
CMD >     Crms : cha=1;                    set and read cha. 1
CMD >     Cmean: cha=1;
CMD >     wait : delay = 150;
CMD >     read : Crms, Cmean;
CMD >     file : store, to_file_# = 2;
CMD >     file : store, to_file_# = 9;
CMD > end: 1;
CMD > ;
CMD > file: close, to_file_# = 1;
CMD > file: close, to_file_# = 2;
CMD > file: close, to_file_# = 9;
CMD > ;
CMD > dos := ANALYSIS <RmsMean0.dat >prn;
CMD > dos := ANALYSIS <RmsMean1.dat >prn;
```

.. END OF HELP ..

APPENDIX B - SOURCE LISTING OF THE CADA-CTA PACKAGE

A: SYSTEM REQUIREMENTS

To use this package, you must have the following:

1: An IBM-PC, an IBM-XT or an IBM-AT (or a compatible).

2: The CADA-CTA package requires approximately 384k of RAM memory.

3: One 360k disk drive is sufficient.

4: DISA Constant Temperature Anemometer and Signal Analysis Equipment.

5: DISA four Axes Traversing Unit.

6: I/O Tech's Serial to IEEE-488 protocol converter.

7: The NATIONAL INSTRUMENTS "GPIBPCIA" interface are required.

Switch setting are as follow: BASE I/O addr :: _____
DMA channel :: _____
Interrupt line:: _____

B: TO START THE CADA-CTA PACKAGE (from floppy drive)

1: Insert the supplied diskette into the BOOT drive. (A:).

2: Turn the computer 'ON'.
This supplied diskette contains the necessary DRIVERs required by this package to run correctly if the above hardware requirements are met.

3: The AUTOEXEC batch will bring up the CADA-CTA software.

C: Installing the CADA-CTA package to the hard disk drive

1: Boot system.

2: Insert the supplied diskette into drive A:.

3: Make a directory in the hard drive with the name CADA, change directory to CADA and copy all files from A: to CADA.

```
C:> md \CADA-CTA
C:> cd \CADA-CTA
C:> copy a:*.* .
```

D: Execute the CADA-CTA program from the hard disk system

1: Type in the following :

```
C:> cd \CADA-CTA
C:> CADA-CTA
```

C: FILES SUPPLIED

The entire package is supplied on a single diskette which contains the executable files as well as some example command files. You will find the following files on the diskette supplied.

AUTOEXEC.BAT	- Batch file used to invoke the CADA-CTA package.
CONFIG .SYS	- This file must be present when BOOTING the system.
CADA-CTA.COM CADA_OVL.CHN	- Executable files for the CADA-CTA package.
CADA-CTA.DEF	- Window definition file is used internally to maintain windows information.
CADA-CTA.HLP	- On-Line help file.
GPIB .COM	- Communication driver for the NATIONAL INSTRUMENTS GPIB-PC2A interface.
README .CTA	- Contains this file.
* .CMD	- Example command file which can be executed by the CADA-CTA package.

FILES REQUIRED TO DO CADA DEVELOPMENT WORK

Text files :

readme	cta	: Include - instruction for software installation - Source file descriptions

Files required to run CADA-CTA program

cada-cta com	: CADA-CTA main program
cada_ovl chn	: additional program required for CADA-CTA
cada-cta hlp	: On-Line help file
cada-cta def	: Window definition file

Sample command files for CADA-CTA program

Example command files :

fil4	cmd
init	cmd
sample	cmd
_0trav	cmd
_init	cmd

Programs needed to do CADA development work

Compiler and other programs

TDEBUG	COM	: Turbo Pascal debuger
TURBO	MSG	: Turbo Pascal compiler message file
BIGMAKE	COM	: Make utility for turbo pascal
BTURBO	COM	: Special version of turbo which supports the make utilities
SHELLGEN	COM	: Generate turbo program bigger than 64K
BIGTURBO	VAR	: files required by SHELLGEN, BIGMAKE & BTURBO.
FARCTRL	INC	
LOADMOD	INC	

Intstruction to compile the CADA-CTA package

```
shellgen -B -A -S cada-cta
bigmake  -A -C -M cada-cta
```

Files required to create CADA-CTA.EXE

make	bat	: Batch / Make files for compiling CADA-CTA.COM
td	bat	: Batch file used to debug CADA-CTA.COM

Source files for the window management module (CADA-CTA.COM)

Binary files used with the following source files

dd-ni488	bin	: Device driver for National Instrument's IEEE-488 card
ram-page	bin	: Driver for the windowing interface
spawn4ah	bin	: DOS Call interface
spawn4bh	bin	: DOS Call interface

Pascal source code for CADA-L.COM

Main program :

cada-cta	pas	: Command Interpreter and Window management module

Command Interpretation modules :

cli-cta	drv	: Implementation of '
cli-dos	drv	: Implementation of 'DOS' command
cli-dwe	drv	: Implementation of 'DO', 'Wait' & 'Execute' command
cli-edh	drv	: Implementation of 'EDIT' command
cli-file	drv	: Implementation of 'FILE' command
cli-prob	drv	: Implementation of 'PROBE' command
cli-sae	drv	: Implementation of commands for the SAE devices
cli-sam	drv	: Implementation of 'SAM' command
cli-tra	drv	: Implementation of 'TRAVERSE' command

Device drivers modules :

dd-cta	drv	: PC to CTA communication routines
dd-gpib	drv	: Interface module for the GPIB / IEEE-488 protocol
dd-ni488	drv	: National Instruments IEEE-488 interfacing code
dd-probe	drv	: PC to stepper motor controller
dd-sae	drv	: PC to SAE communication routines
dd-sam	drv	: PC to SAM communication routines
dd-trav	drv	: Convert high level command to IEEE-488 calls

Code required by Shellgen & Bigmake :

BIGTURBO	VAR	: Files required by SHELLGEN, BIGMAKE & BTURBO.
FARCTRL	INC	
LOADMOD	INC	

Utilities modules :

spawn4ah	asm	: DOS CALL interface
spawn4bh	asm	: DOS CALL interface
er-stack	inc	: General debugging routines
get-in	inc	: Keyboard routines
library	inc	: General utilities routines
ram-page	inc	: Windowing interface routines
rwindow	inc	: Windowing interface routines
spawn	inc	: DOS program spawning routines
string	inc	: String utilities

Directory of \CADA-CTA

spawn4ah	asm	2,304	11-13-86	11:06a
spawn4bh	asm	4,992	11-13-86	11:07a
make	bat	500	10-13-88	12:16p
td	bat	56	11-17-86	11:17a
dd-ni488	bin	343	7-14-86	3:35p
ram-page	bin	5,700	4-09-86	11:13a
spawn4ah	bin	58	11-13-86	11:12a
spawn4bh	bin	111	11-13-86	11:12a
cada_ovl	chn	26,891	10-13-88	1:13p
file	cmd	753	11-25-86	1:26p
init	cmd	23	1-09-87	9:40a
sample	cmd	1,617	11-25-86	1:29p
_0trav	cmd	665	11-25-86	8:57a
_init	cmd	601	11-25-86	9:04a
cada-cta	com	57,843	10-13-88	1:13p
readme	cta	7,075	11-11-88	4:33p
cada-cta	def	1,240	10-05-88	10:58p
cli-cta	drv	4,992	11-24-86	9:20a
cli-dos	drv	2,763	1-20-87	12:52p
cli-dwe	drv	9,250	11-11-88	3:41p
cli-edh	drv	3,024	11-11-88	3:42p
cli-file	drv	17,782	11-11-88	3:42p
cli-prob	drv	1,411	11-17-86	3:12p
cli-sae	drv	2,820	11-09-86	8:53p
cli-sam	drv	3,467	11-11-88	3:45p
cli-tra	drv	5,519	11-24-86	11:48a
dd-cta	drv	10,592	11-27-86	12:25p
dd-gpib	drv	6,995	10-13-88	12:09p
dd-ni488	drv	7,463	10-13-88	12:06p
dd-probe	drv	6,512	10-13-88	12:14p
dd-sae	drv	6,195	12-22-86	9:32a
dd-sam	drv	13,010	12-22-86	9:19a
dd-trav	drv	5,221	11-27-86	12:32p
cada-cta	hlp	21,255	10-31-88	4:09p
er-stack	inc	4,447	11-07-86	12:19a
farctrl	inc	1,315	9-05-86	12:00a
get-in	inc	3,668	11-11-88	3:48p
library	inc	4,490	11-25-86	11:41a
loadmod	inc	16,203	9-05-86	12:00a
ram-page	inc	1,399	10-05-88	10:19p
rwindow	inc	16,076	11-11-88	3:49p
spawn	inc	8,011	10-13-88	12:07p
string	inc	2,909	11-88	3:50p
cada-cta	pas	28,851	11-11-88	3:51p
bigturbo	var	7,633	9-05-86	12:00a

47 files found 1,251,328 bytes free

```
page            55,132
title           DOS4AH (pp_to_release: integer): integer
code            segment
                assume          cs:code

;       function called from Turbo Pascal, which releases PP_TO_RELEASE
;       paragraphs of memory from the current memory block using the DOS
;       function 4AH.  If this is possible, the the Turbo stack is
;       relocated to the high end of the new allocation and function
;       result is zero.  otherwise the DOS error code is returned.

;       Pascal declaration:
;       FUNCTION DOS4AH(PP_TO_RELEASE: INTEGER): INTEGER;

arguments       struc                   ; template to arguments.
save_bp         dw      ?               ; saved BP register.
ret_adr         dw      ?               ; return address (near call).
;-------------------------------------------------------------------------
pp_to_release   dw      ?               ; integer # of paragraphs to release.
;-------------------------------------------------------------------------
arguments       ends

dos4AH          proc    near
                push    bp
                mov     bp,sp           ; establish access to arguments
                push    ds              ; save Turbo's data segment.

                mov     ax,cs           ; calculate the paragraph size
                mov     bx,ss           ; of the current block as
                add     bx,1000H        ; SS+1000H minus code segment
                sub     bx,ax           ; address. Then calculate new size
                                        ; for memory block.
                sub     bx,[bp].pp_to_release
                mov     es,ax           ; segment address of current
                mov     ah,4AH          ; memory block.
                int     21H             ; ask DOS to deallocate the space.
                mov     ah,0
                jc      exit            ; was it possible to release the memory?

                mov     ax,ss           ; Yes:  set up the relocate stack
                mov     ds,ax           ; DS:SI is pointer to old stack
                sub     ax,[bp].pp_to_release
                mov     es,ax           ; ES:DI is the pointer to the new stack.
                mov     bx,sp
                mov     di,bx
                mov     si,bx

                mov     cx,sp           ; two's complement of SP is the stack
                neg     cx              ; size.
                cld                     ; move cx bytes from
                rep     movsb           ; DS:SI to ES:DI ...
                                        ; starting with the lowest byte
                mov     ss,ax           ; change the SS register ...
                xor     ax,ax           ; and set the error code to 0.

exit:           pop     ds
                pop     bp
                ret     4               ; Pop argument and "function result"

dos4AH          endp

code            ends

                end
```



```
page            55,132
title           DOS4BH (var program_name, parameter_string): integer
code            segment
                assume          cs:code

; FUNCTION: called from Turbo Pascal, executes .COM
; or .EXE programs using the DOS function 4BH.
; Preserves the stack registers, returning the
; error code 0 if successfull or the DOS error code
; if the execute fails.

; Pascal declaration:
; FUNCTION DOS4BH (var program_name, parameter_string): integer

arguments       struc                   ; template to arguments.
save_bp         dw      ?               ; saved BP register.
ret_adr         dw      ?               ; return address (near call).
;-------------------------------------------------------------------------
param_str       dd      ?               ; DWORD pointer to param. string
prog_name       dd      ?               ; DWORD pointer to ASCIIZ
                                        ; program name.
;-------------------------------------------------------------------------
arguments       ends

dos4BH          proc    near
                jmp     short exec_code
;-------------------------------Load control block----------------------
lcb_envir       dw      0               ; segment address of environment
lcp_ps_o        dw      ?               ; offset of parameter string.
lcb_ps_s        dw      ?               ; segment of parameter string.
lcb_fcb1        dd      0               ; DWORD pointer to FCB1.
                                        ; (5CH in PSP).
lcb_fcb2        dd      0               ; DWORD pointer to FCB2
                                        ; (6CH in PSP)
;-------------------------------Local variables-------------------------
ss_save         dw      ?               ; saved stack segment.
sp_save         dw      ?               ; saved stack pointer.
;-----------------------------------------------------------------------
exec_code:      push    bp
                mov     bp,sp           ; establish access to arguments.
                push    ds              ; save the Turbo data stack.
                call    relo1           ; set BX to relocation
                                        ; factor: true offset ...
relo1:          pop     bx              ; of relo1 minus local offset
                sub     bx,offset relo1

                les     di,[bp].param_str ; get offset:segment
                                        ; of parameter string.
                mov     ax,es
                mov     cs:lcb_ps_s[bx],ax ; save parameter
                                        ; string segment ...
                mov     cs:lcp_ps_o[bx],di ; ... and ofset in LCB.
                mov     ax,ss           ; save SS and SP registers
                mov     cs:ss_save[bx],ax ;
                mov     ax,sp           ;
                mov     cs:sp_save[bx],ax ;
                push    cs              ; set up ES:BX as pointer
                                        ; to the LCB ...
                pop     es              ;
                add     bx,offset lcb_envir
                lds     dx,[bp].prog_name ; and DS:DX as pointer
                                        ; to the ASCIIZ ...
                inc     dx              ; ... program name.
                mov     ax,4B00H        ; DOS EXEC FUNC: al=0, ah=4BH
                int     21H             ; let her rip.

                sti                     ; disable interrupts while
                                        ; stack messed up
                jc      run_error       ; sucessful execution ?
                xor     al,al           ; Yes, set error code = 0.
```

```
run_error:  xor     ah,ah           ;
            call    relo2           ; recalculate relocation factor ...
relo2:      pop     bx              ;
            sub     bx,offset relo2 ;
            mov     dx,cs:ss_save[bx] ; ... and restore the stack.
            mov     ss,dx           ;
            mov     dx,cs:sp_save[bx] ;
            mov     sp,dx           ;
            cli                     ; interrupts back on ... all done!

            pop     ds
            pop     bp
            ret     10              ; pop arguments and "function result"
dos4BZ      endp
code        ends
            end
```



```
Var _Cms : record
             dev   : byte;
             cha   : byte;
             tc    : real;
             OL    : boolean;
             data  : real;
           end;

Var  _Cmean : record
              dev    : byte;
              chA    : byte;
              tc     : real;
              filter : boolean;
              OL     : boolean;
              data   : real;
            end;

Type _cond_cha_rec = record
                       gain : real;
                       lp   : real;
                       hp   : real;
                       disp : boolean;
                     end;

Var  _Cond : record
             dev  : byte;
             ch   : byte;
             cha  : array[0..2] of _cond_cha_rec;
           end;


procedure init_cli_cta;
  var i : byte;

  begin
    with _Cms do begin
      dev := 0;  chA := 0;  tc := 0.0;  data := 0.0;
      OL  := false;
    end;

    with _Cmean do begin
      dev := 0;  chA := 0;  tc := 0.0;  filter := false;  data := 0.0;
      OL  := false;
    end;

    _Cond.dev := 0;
    _Cond.ch  := 0;

    for i:=0 to 2 do begin
      with _Cond.cha[i] do begin
        gain := 0.0;  lp := 0.0;  hp := 0.0;
        disp := false;
      end;
    end;

    _Cond.cha[0].disp := true;
  end;


Procedure cli_CMS (token   : st3;              { sub command token  }
                   Var buf : st128);           { command line       }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

  BEGIN
   Push_Msg (' Proc: cli_Cta-RMS');
    err := 0;
```

```
    IF (token='CEA') or (token='CES') or (token='TC') or (token='DEV')
       then begin
       IF ( FoundData( buf, data ) ) then begin
          VAL (data, r_data, v_code);
          IF (v_code = 0) then begin
             IF      (token = 'CEA') then _Cmss.cha := TRUNC(r_data)
             ELSE IF (token = 'TC' ) then _Cmss.tc  := r_data
             ELSE if (token = 'DEV') then _Cmss.dev := TRUNC(r_data)
          end
          ELSE  err := 1;
       end
       ELSE err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
  Pop_Msg;
 END; { cli_Cta-RMS }
(.pa)

Procedure cli_CMEAN (token   : st3;              { sub command token }
                     Var buf : st128);           { command line      }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

 BEGIN
  Push_Msg (' Proc: cli_Cta-MEAN');
    err := 0;
    IF (token='CEA') or (token='TC' ) or (token='DEV')
       then begin
       IF ( FoundData( buf, data ) ) then begin
          VAL (data, r_data, v_code);
          IF (v_code = 0) then begin
             IF      (token = 'CEA') then _Cmean.cha := TRUNC(r_data)
             ELSE IF (token = 'TC' ) then _Cmean.tc  := r_data
             ELSE if (token = 'DEV') then _Cmean.dev := TRUNC(r_data)
          end
          ELSE  err := 1;
       end
       ELSE  err := 2;
    end
    ELSE if (token = 'FIL') then begin
       IF ( FoundData( buf, data ) ) then begin
          data3 := data;
          if (data3 = 'OFF') then
             _Cmean.filter := false
          else
             _Cmean.filter := true;
       end
       else err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
  Pop_Msg;
 END; { cli_MEAN }
(.pa)

Procedure cli_COND (token   : st3;               { sub command token }
                    Var buf : st128);            { command line      }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

 BEGIN
  Push_Msg (' Proc: cli_COND');
    err := 0;
```



```
    IF (token='CEA') or (token='GAI') or (token='LP') or (token='HP')
       or (token='DEV') then begin
       IF ( FoundData(buf, data) ) then begin
          VAL (data, r_data, v_code);
          IF (v_code = 0) then begin
             IF      (token = 'CEA') then _cond.ch  := TRUNC(r_data)
             ELSE if (token = 'DEV') then _cond.dev := TRUNC(r_data)
             else begin
               with _cond.cha[_cond.ch] do begin
                 IF      (token = 'GAI') then gain:= r_data
                 ELSE IF (token = 'LP' ) then lp  := r_data
                 ELSE IF (token = 'HP' ) then hp  := r_data
               end;
             end;
          end
          ELSE  err := 1;
       end
       ELSE err := 2;
    end
    ELSE if (token = 'DIS') then begin
       IF ( FoundData(buf, data) ) then begin
          data3 := data;
          if (data3 = 'OFF') then
             _Cond.cha[_cond.ch].disp := false
          else
             _Cond.cha[_cond.ch].disp := true;
       end
       else err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
  Pop_Msg;
 END; { cli_COND }
(.pa)
```

```
{*------------------------------------------------------------------*}
{ Oct 11, 86                  "get current dir & path name"          }
{*------------------------------------------------------------------*}

{ ---------------------------------------------------------------- }
procedure release_heap (pp_to_release: integer);
  label finish;
  var   memory_segment : integer;

  begin
    if dos(4Ah (pp_to_release, memory_segment) = 0 then begin
      if dos_error_check (dos(4Ah (memory_segment)) then goto finish;
    end
    { ------------- if not, then reduce the size of the current allocation }
    else begin
      if dos_error_check (dos(4Ah (pp_to_release)) then goto finish;
    end;

    finish:
  end;


procedure init_dos (block_to_reserve : integer);
  var mem : integer;
  begin
    { --------- reserve internal storage and release the rest from the HEAP }
    release_heap (block_to_reserve);
  end;


type dos_string = string[128];

procedure cli_dos (var parameter_string : dos_string);
  label finish;
  var   program_name,  prompt_str       : ascii;
        mem_avail_in_program, dum_int   : integer;
        opt_loc, par_loc                : integer;
        pause                           : boolean;
        dum_key                         : char;

  begin
    { ------------------------------ terminate if can't get comspec name }
    if get_comspec (program_name) then begin
      writeln (bell, 'Missing COMSPEC');
      goto finish;
    end;

    { ------------------------------------- analysis command string }
    pause := true;

    opt_loc := pos ('-p', parameter_string);
    dum_int := pos ('-P', parameter_string);
    if ( (dum_int > 0) and (dum_int < opt_loc) ) then
      opt_loc := dum_int;

    par_loc := pos (':-', parameter_string);
    if (opt_loc = 0) then
      pause := true
    else if ( (par_loc = 0) and (opt_loc > 0) ) then
      pause := false
    else if ( (par_loc > 0) and (opt_loc < par_loc) ) then
      pause := false;

    if par_loc = 0 then
      parameter_string := program_name
    else begin
      delete (parameter_string, 1, par_loc+1);
      parameter_string := parameter_string;
    end;
```



```
    parameter_string := '/c '+ parameter_string;

    { ---------- }
    writeln ('DOS > ', parameter_string);
    {----------}

    parameter_string [length (parameter_string)+1] := #13;

    if dos_error_check (dos(4Bh (program_name, parameter_string)) then
      writeln ('Command Cancelled.');

    finish:
      if (pause) then begin
        gotoXY (49,25);
        write  (' Press RETURN to continue .....');
        read   (dum_key);
      end;
  end;
```

```
{ - - - - DO LOOP INTERPRETER ROUTINES - - - - - - - - - - - - - - - - - }
TYPE  do_ptr     = ^do_record;
      do_record  = RECORD
                     text   : st128;
                     next   : do_ptr;
                   END;
      do_stru    = RECORD
                     lab    : integer;
                     ctr    : integer;
                     first  : do_ptr;
                   END;

VAR   _do_rec          : do_ptr;
      _do              : array [0..9] of do_stru;
      _do_flg,
      _do_stored_flg : boolean;
      _do_level      : integer;
      _do_first,
      _do_last,
      _do_rec_ptr    : do_ptr;
      _do_cur_lab    : integer;
{-------------------------}

Procedure Dispose_All_Loop;
  VAR   temp_ptr : do_ptr;
  BEGIN
     _do_flg := FALSE;
     While (_do[0].first <> nil) do begin
        temp_ptr := _do[0].first;
        _do[0].first := temp_ptr^.next;
        DISPOSE (temp_ptr);
     end;
  END;

{.pa}

Function cli_STORE_DO (Var buf : st128) : boolean;
  VAR  data           : st10;
       buf128         : st128;
       v_code, I_data : integer;
       token3         : st3;
       label_begin,
       label_end      : integer;
       temp_ptr       : do_ptr;
       error          : boolean;

  BEGIN
   Push_msg (' Proc: cli_Store_DO');
     _do_flg        := TRUE;
     _do_level      := -1;
     _do_cur_lab    := -1;
     _do_stored_flg := FALSE;

     NEW (_do_rec);
     _do_rec^.text := buf;
     _do_rec^.next := nil;
     _do_first     := _do_rec;
     _do_last      := _do_rec;

     error := false;

     buf128 := copy(buf, 4, length(buf)-3);
     IF (FoundData (buf128, data)) then begin
        VAL (data, I_data, v_code);
        IF (v_code = 0) then
           label_begin := I_data
        ELSE begin
           AbortCLI ('invalid LABEL entry');
           error := true;
        end;
```



```
        end
        ELSE begin
           AbortCLI ('no LABEL are specified');
           error := true;
        end;

        if (not error) then begin
          REPEAT
            Get_Input (' DO', buf);
            NEW (_do_rec);
            _do_rec^.text  := buf;
            _do_rec^.next  := nil;
            _do_last^.next := _do_rec;
            _do_last       := _do_rec;

            token3 := buf;
            IF (token3 = 'END') then begin
              buf128 := COPY(buf, 5, LENGTH(buf)-4);
              IF (FoundData (buf128, data)) then begin
                VAL (data, I_data, v_code);
                IF (v_code = 0) then
                  label_end := I_data
                ELSE begin
                  AbortCLI ('invalid LABEL entry');
                  error := true;
                end;
              end
              ELSE begin
                AbortCLI ('no LABEL are specified');
                error := true;
              end;
              IF (label_begin = label_end) then
                _do_stored_flg := TRUE;
            end;
          UNTIL ( _do_stored_flg or error);

          printfln ('');

          _do_rec_ptr := _do_first;

          { ----
          IF (debug_488 > 2) then begin
            temp_ptr := _do_first;
            REPEAT
              IF (temp_ptr <> nil) then begin
                printfln (temp_ptr^.text);
                temp_ptr := temp_ptr^.next;
              end;
            UNTIL (temp_ptr = nil);
          end;
          ---- }
        end;

        if error then begin
          Dispose_all_loop;
          cli_store_do := false;
        end
        else
          cli_store_do := true;
     Pop_msg;
    END; { cli_STORE_DO }

{.pa}

 Procedure cli_Begin_DO (buf : st128);
   VAR  com, data     : st10;
        I_data, v_code : integer;
        err            : byte;
   BEGIN
    Push_msg (' Proc: cli_Begin_DO');
```

```
    err := 0;
    IF (FoundData (buf, data)) then begin
       VAL (data, I_data, v_code);
       IF (v_code = 0) then begin

          IF (_do_cur_lab <> I_data) then begin
             _do_level := _do_level + 1;
             _do[_do_level].lab := I_data;
             _do_cur_lab := I_data;
             _do[_do_level].first := _do_rec_ptr;

             IF (FoundCom (buf, '=', com)) then begin
                IF (com[1] = '?') then begin

                   IF (FoundData (buf, data)) then begin
                      VAL (data, I_data, v_code);
                      IF (v_code = 0) then
                         _do[_do_level].ctr := I_data
                      ELSE err := 1;
                   end
                   ELSE err := 2;
                end
                ELSE err := 3;
             end
             ELSE err := 2;
          end;
       end
       ELSE AbortCLI ('invalid DO LOOP LABEL');
    end
    ELSE AbortCLI ('missing LABEL');
    { ----
    IF (debug_488 >4) then
       writeln (' stored  lab & times= ', _do[_do_level].lab:9, _do[_do_level].ctr:9);
    ---- }
    DisplayError (err);
  Pop_msg;
END; { cli_BEGIN_DO }

{.pa}

Procedure cli_End_DO (buf : st128);
  VAR  com, data      : st10;
       I_data, v_code : integer;
       temp_ptr       : do_ptr;
  BEGIN
   Push_msg (' Proc: cli_End_DO');

    IF (FoundData (buf, data)) then begin
       VAL (data, I_data, v_code);
       IF (v_code = 0) then begin

          IF (_do_cur_lab = I_data) then begin
             _do[_do_level].ctr := _do[_do_level].ctr - 1;

             IF (_do[_do_level].ctr <= 0) then begin
                _do_level := _do_level - 1;

                IF (_do_level >= 0) then begin
                   _do_cur_lab := _do[_do_level].lab;
                end
                ELSE
                   Dispose_All_Loop;
             end
             ELSE _do_rec_ptr := _do[_do_level].first;
          end
          ELSE AbortCLI ('DO: label mismatch');
       end
       ELSE AbortCLI ('invalid DO LOOP LABEL');
    end
    ELSE AbortCLI ('missing LABEL for the END statement');
```



```
   Pop_msg
  END; { cli_END_DO }

{.pa}

Procedure cli_WAIT (token   : st3;              { sub command token }
                    Var buf : st128);           { command line      }
  Var   data, time              : st10;
        current_time, wait_time : string[8];
        time2                   : st2;
        r_data                  : real;
        v_code, err, i          : integer;
        key                     : char;
        r_hr,r_min,r_sec        : integer;
        v_code1,v_code2,v_code3 : integer;
  BEGIN
   Push_Msg (' Proc: cli_wait');
     err := 0;
     IF (token = 'KEY') or (token = 'TIM') or (token = 'DEL' ) then begin

        IF (token = 'KEY') then begin
          i := whereY;
          gotoXY(1,i);      clrEOL;
          TextColor(blink); printf (' Press any key to continue ');  NormVideo;
          read (kbd, key);
          Open_user_window;
          gotoXY(1,i);      clrEOL;
        end
        ELSE IF ( FoundData(buf, data) ) then begin
           IF (token = 'TIM') then begin
              r_hr := -1;   r_min := -1;   r_sec := -1;
              IF (length(data) = 8) then begin
                 wait_time :=   COPY(data, 1, 2) + ':'
                              + COPY(data, 4, 2) + ':'
                              + COPY(data, 7, 2);
                 FOR i:=1 to 8 do
                    IF (wait_time[i] = ' ') then   wait_time := '0';
              end;
              IF ( (wait_time < '00:00:00') or (wait_time > '23:59:59') ) then
                 AbortCLI ('invalid time*spec.: HH:MM:SS');

              REPEAT
                 current_time := Get_Time;
                 update_time_date (false);
              UNTIL (keypressed) or (current_time >= wait_time);
              printfln ('');
           end
           ELSE IF (token = 'DEL') then begin
              VAL (data, r_data, v_code);
              IF (v_code = 0) then
                 DELAY ( TRUNC( r_data * 1000 ) )
              ELSE  err := 1;
           end
           ELSE err := 1;
        end
        ELSE  err := 2;
     end
     ELSE  err := 3;
     DisplayError (err);
   Pop_msg;
  END; { cli_WAIT }

{.pa}

Procedure cli_EXECUTE_1 (token   : st3;         { sub command token }
                         Var buf : st128);      { command line      }
  Var   data          : st10;
        r_data        : real;
        v_code, err   : integer;
```

```
  BEGIN
   Push_Msg (' Proc: cli_EXECUTE_1');
     err := 0;
     IF (token = 'FIL') or (token = 'USE') or (token = 'FAR') then begin
        IF ( FoundData(buf, data) ) then begin
           CASE (token[1]) of
             'F' : begin
                     ASSIGN (cmd_file[cmd_num +1], data);
                     {$I-}  RESET (cmd_file[cmd_num +1]);  {$I+}
                     IF (IOresult <> 0) then
                        printfln (' CANNOT OPEN CMD_FILE: ' + data)
                     ELSE begin
                        cmd_num := cmd_num + 1;
                        cmd_file_exist[cmd_num] := TRUE;
                     end;
                   end;
             ELSE  err := 1;
           end;
        end
        ELSE  err := 2;
     end
     ELSE  err := 3;
     DisplayError (err);
   Pop_Msg;
  END; { cli_EXECUTE_1 }

{.pa}
```



```
Procedure cli_Enable (device  : ValidCmd_t;
                      on_off  : boolean);
   VAR found : boolean;
   BEGIN
      IF (device = c_all) then  printfln ('');

      IF (on_off) then
         printfln ('Device: "' + ValidCmd_s[integer(device)] + '" enabled')
      ELSE
         printfln ('Device: "' + ValidCmd_s[integer(device)] + '" disabled');

      With enable DO begin
         CASE device of
           c_rms   : cta_rms  := on_off;
           c_mean  : cta_mean := on_off;
           c_cond  : cond := on_off;
           s_rms   : sae_rms  := on_off;
           s_mean  : sae_mean := on_off;
           s_corr  : sae_corr := on_off;
           s_sam   : sae_sam  := on_off;
           c_trav  : trav := on_off;
           c_probe : probe:= on_off;
           c_bell  : if (on_off) then begin
                       Bell := ^G;
                       printfln ['Bell enabled');
                     end
                     else begin
                       Bell := #15;
                       printfln ('Bell disabled');
                     end;
           c_all   : Begin
                       cli_enable (c_rms,   on_off);
                       cli_enable (c_mean,  on_off);
                       cli_enable (c_cond,  on_off);
                       cli_enable (s_rms,   on_off);
                       cli_enable (s_mean,  on_off);
                       cli_enable (s_corr,  on_off);
                       cli_enable (s_sam,   on_off);
                       cli_enable (c_trav,  on_off);
                       cli_enable (c_probe, on_off);
                     end;
         end;
      end;
   END;

{.pa}

Procedure cli_DEBUG (level_str : st3;
                     var buf   : st128);
  VAR i, v : integer;
      data : st10;
  BEGIN
     IF (NOT FoundData (buf, data)) then
        AbortCLI ('incorrect debug level <0..9>')
     ELSE begin
        VAL (data, i, v);
        debug_488 := i;
        writeln ('Debug level changed to ', debug_488);
     end;
  END; { cli_DEBUG }


Procedure cli_HELP ;
   VAR help_file : text;
       ch        : char;
       buf       : st128;
   BEGIN
     if (_help_loaded) then
       Change_Screen (help, Scroll)
     else begin
```

```
      ASSIGN (help_file, 'CADA-CTA.ELP');
      {$I-}  RESET(help_file);  {$I+}
      IF (IOresult <> 0) then begin
         printfln ('HELP file does not exist in current directory');
         printfln ('');
      end
      ELSE begin
        _help_loaded := true;
        with W[help] do begin
          TextColor       (foreground);
          TextBackGround  (background);
          Ram_Page        (handle);
          WHILE (not EOF(help_file)) DO begin
            READLN (help_file, buf);
            printfln (buf);
          end;
          CLOSE (help_file);
          Appear_view     (handle);
          Change_Screen  (help, Scroll);
        end;
      end;
    end;
    Open_User_Window;
  END;

{.pa}
```



```
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
VAR data_file : array[0..9] of TEXT;
    file_rec  : array[0..9] of RECORD
                                 device    : char;
                                 open      : boolean;
                                 format    : string[80];
                               END;

Procedure cli_STORE_DATA_FILE (file_num : integer;
                               title    : boolean);        { title or data }
  VAR format_str               : st128;
      form_buf                 : st10;
      date_str, time_str       : string[8];
      i, j, space, chan, v_code : integer;
  BEGIN
   Push_msg (' Proc: cli_STORE_DATA_FILE');
     format_str := file_rec[file_num].format;
     IF (title) then  write (data_file[file_num], '*')
     ELSE  write (data_file[file_num], '>');

     WHILE ( FoundDATA (format_str, form_buf)) do begin
        CASE (form_buf[1]) OF
          'D' : begin                                                 { DATE }
                  date_str := Get_DATE;
                  IF (title) then
                    write (data_file[file_num], '|___Date':9)
                  ELSE
                    write (data_file[file_num], date_str:9);
                end;

          'T' : begin
                  IF (form_buf[2] = 'I') then begin                   { TIME }
                    time_str := Get_TIME;
                    IF (title) then
                      write (data_file[file_num], '|___Time':9)
                    ELSE
                      write (data_file[file_num], time_str:9);
                  end
                  ELSE begin                                          { TRAV }
                    IF (title) then
                      write (data_file[file_num], '|__X_axis':10,
                                                  '___Y_axis':10,
                                                  '___Z_axis':10,
                                                  '___R_axis':10)
                    ELSE
                      write (data_file[file_num], _trav[0].posn:10:4,
                                                  _trav[1].posn:10:4,
                                                  _trav[2].posn:10:4,
                                                  _trav[3].posn:10:4);
                  end;
                end;

          'X' : case (form_buf[2]) of
                'R' : with _Srms do begin
                        i := POS ('=', form_buf);
                        IF (i > 0) then
                          IF (title) then
                            write (data_file[file_num], '|$RMS':6)
                          ELSE
                            write (data_file[file_num], data:6:2)
                        ELSE
                          IF (title) then
                            write (data_file[file_num], '|chA':5,  'chB':4,
                                                        '___TC':6, '_$RMS':6, ' Mode':6)
                          ELSE
                            write (data_file[file_num], cha:5,    chb:4,
                                                        tc:6:1,   data:6:2, mode:6);
                      end;
                'M' : with _Smean do begin
                        i := POS ('=', form_buf);
```

```
            IF (i > 0) then
               IF (title) then
                  write (data_file[file_num], '|__SMEAN':9)
               ELSE
                  write (data_file[file_num], data:9:6)
            ELSE
               IF (title) then
                  write (data_file[file_num], '|ch':4,
                                              '____TC':7,
                                              '___SMEAN':9)
               ELSE
                  write (data_file[file_num], cha:4,
                                              tc:7:1,
                                              data:9:6);
         end;
      'O' : with _Scorr do begin
            IF (title) then
               write (data_file[file_num], '|chX':4,
                                           'chY':4,
                                           '___Idelay':9,
                                           '___Sdelay':9,
                                           '___Tfactor':9,
                                           'Status':7,
                                           '_Error':7)
            ELSE
               write (data_file[file_num], chX:4,
                                           chY:4,
                                           Idelay:9:4,
                                           Sdelay:9:4,
                                           HEX(Status):7,
                                           HEX(err_code):7);
         end;
      'A' : with _Saam do begin
            i := POS ('=', form_buf);
            IF (title) then begin
               write (data_file[file_num], '|');
               if (i = 0) then begin
                  write (data_file[file_num], 'Trig':5,
                                              'Proc':5,
                                              'Test':5,
                                              '__ACCTIM':9,
                                              '__Real_T':9,
                                              'Remain_T':9);
               end;
               write (data_file[file_num],    '__Status':9);
               for j:=0 to 7 do
                  write (data_file[file_num], ' __SAM[', j:1, ']');
            end
            ELSE begin
               write (data_file[file_num], ' ');
               if (i = 0) then begin
                  write (data_file[file_num], Trig:5,
                                              Proc:5,
                                              Test:5,
                                              ACCTIM:9:4,
                                              RealT:9:4,
                                              RemainT:9:4);
               end;
               write (data_file[file_num],    HEX(Status):9);
               for j:=0 to 7 do
                  write (data_file[file_num], Result[j]:9:4);
            end;
         end;
      'P' : begin                                              [ Space ]
            i := POS ('=', form_buf);
            IF (i = 0) then
               write (data_file[file_num], ' ')
            ELSE begin
               VAL (form_buf[i + 1], space, v_code);
               IF ( (v_code <> 0) or (space < 0) ) then
```



```
                  AbortCLI ('invalid space selection')
               ELSE
                  write (data_file[file_num], ' ':space);
            end;
         end;
      end;

   'C' : case (form_buf[2]) of
         'R' : with _Crms do begin
               i := POS ('=', form_buf);
               IF (i > 0) then
                  IF (title) then
                     write (data_file[file_num], '|CRMS':6,
                                                 '___OL':6)
                  ELSE
                     write (data_file[file_num], data:6:2,
                                                 OL:6)
               ELSE
                  IF (title) then
                     write (data_file[file_num], '|chA':5,
                                                 '___TC':6,
                                                 '___OL':6,
                                                 '_CRMS':6)
                  ELSE
                     write (data_file[file_num], cha:5,
                                                 tc:6:1,
                                                 OL:6,
                                                 data:6:2);
            end;
         'M' : with _Cmean do begin
               i := POS ('=', form_buf);
               IF (i > 0) then
                  IF (title) then
                     write (data_file[file_num], '|__CMEAN':9,
                                                 '__OL':6)
                  ELSE
                     write (data_file[file_num], data:9:6,
                                                 OL:6)
               ELSE
                  IF (title) then
                     write (data_file[file_num], '|ch':4,
                                                 '____TC':7,
                                                 '____OL':7,
                                                 '____FIL':7,
                                                 '___CMEAN':9)
                  ELSE
                     write (data_file[file_num], cha:4,
                                                 tc:7:1,
                                                 OL:7,
                                                 filter:7,
                                                 data:9:6);
            end;

      'O' : begin                                              [ Cond ]
            i := POS ('=', form_buf);
            IF (i = 0) then
               AbortCLI ('missing channel #')
            ELSE begin
               VAL (form_buf[i+1], chan, v_code);
               IF ( (v_code <> 0) or (chan < 0) or (chan > 2) ) then
                  AbortCLI ('invalid channel #')
               ELSE begin
                  IF (title) then
                     write (data_file[file_num], '|Gain':6, chan:1,
                                                 '___LP':6, chan:1,
                                                 '___EP':6, chan:1)
                  ELSE begin
                     with _Cond.cha[chan] do begin
                        write (data_file[file_num], gain:6:1,
                                                    lp:6:1,
```

```
                                                        hp(6:1));
                                end;
                             end;
                          end;
                       end;
                    end;
              end;

        'P' : begin
                 IF (title) then                                    { Probe }
                    write (data_file[file_num], '|Probe':7)
                 ELSE
                    write (data_file[file_num], _probe_ang:7:1);
              end;

        end; { case }
     end; { while }

     IF (title) then
        writeln (data_file[file_num], '|')
     ELSE
        writeln (data_file[file_num]);

     IF (file_rec[file_num].device = 'C') then
        writeln;

   Pop_msg;
  END; { cli_STORE_DATA_FILE }

{.pa}

Procedure cli_FILE  (token   : st3;          { Format, Open, Close or Store }
                     VAR buf : st128);       { sub commands                 }
  VAR istart, iend, ilen   : integer;
      i_data, v_code       : integer;
      data                 : st10;
      format_str           : st128;
      file_num, i          : integer;
      file_name            : string[30];
      file_n3              : st3;
  BEGIN
     Push_msg (' Proc: cli_FILE');
        CASE token[1] OF
        'O' : begin                                                 { Open }
                 iend := POS (',', buf);
                 IF (iend = 0) then
                    AbortCLI ('File Name not found')
                 ELSE begin
                    file_name := COPY (buf, 1, iend -1);

                    DELETE (buf, 1, iend);

                    IF (buf[1] <> 'T') then
                       AbortCLI ('Expecting "TO FILE=<file_num>')
                    ELSE begin
                       iend := POS ('=', buf);
                       DELETE (buf, 1, iend);
                       IF (not FoundData (buf, data)) then
                          AbortCLI ('File_number not found')
                       ELSE begin
                          VAL (data, file_num, v_code);
                          IF (v_code <> 0) then
                             AbortCLI ('invalid File Number')
                          ELSE begin
                             file_rec[file_num].open := TRUE;

                             FOR i := 1 to iend-1 do
                                file_name[i] := UPCASE (file_name[i]);
                             file_n3 := file_name;
                             IF (file_n3 = 'CON') then
```



```
                                file_rec[file_num].device := 'C'
                             ELSE IF (file_n3 = 'PRN') then
                                file_rec[file_num].device := 'P'
                             ELSE
                                file_rec[file_num].device := 'F';

                             ASSIGN (data_file[file_num], file_name);
                             REWRITE (data_file[file_num]);
                             { ----
                             IF (debug_488 > 4) then begin
                                writeln ('file name is: ', file_name);
                                writeln ('file num. is: ', file_num);
                                writeln ('open status : ', file_rec[file_num].open);
                             end;
                             ---- }
                          end
                       end;
                    end;
                 end;
        'F' : begin                                                 { Format }
                 istart := POS ('(', buf);
                 iend := POS (')', buf);
                 ilen := iend - istart -1;
                 IF (ilen > 0) then
                    format_str := COPY (buf, istart+1, ilen);
                 DELETE (buf, 1, iend +1);
                 IF (buf[1] <> 'T') then
                    AbortCLI ('Expecting "TO FILE=<file_num>" ')
                 ELSE begin
                    iend := POS ('=', buf);
                    data := buf[iend+1];
                    VAL (data, i_data, v_code);
                    IF (v_code <> 0) then
                       AbortCLI ('invalid File Number')
                    ELSE begin
                       file_num := i_data;
                       IF (not file_num in [0..9]) then
                          AbortCLI ('File Number too (large or small)');
                    end;
                 end;
                 IF (file_rec[file_num].open = FALSE) then begin
                    { ----
                    IF (debug_488 > 4) then begin
                       writeln ('file num. is ', file_num);
                       writeln ('open status =', file_rec[file_num].open);
                    end;
                    ---- }
                    AbortCLI ('File is not currently open');
                 end
                 ELSE begin
                    file_rec[file_num].format := format_str + ';';
                    cli_STORE_DATA_FILE (file_num, TRUE);
                 end;
              end;
        'S' : begin                                                 { Store }
                 IF (not FoundData (buf, data)) then
                    AbortCLI ('File_number not found')
                 ELSE begin
                    VAL (data, file_num, v_code);
                    IF (v_code <> 0) then
                       AbortCLI ('invalid File Number')
                    ELSE begin
                       { ----
                       IF (debug_488 > 4) then
                          writeln ('file num. is: ', file_num);
                       ---- }
                       IF (file_rec[file_num].open = FALSE) then
                          AbortCLI ('file is not currently open')
                       ELSE
```

```
                    cli_STORE_DATA_FILE (file_num, FALSE);
                end;
              end;
            end;
      'C' : begin                                          { Close }
              IF (not FoundData (buf, data)) then
                AbortCLI ('File_number not found')
              ELSE begin
                VAL (data, file_num, v_code);
                IF (v_code <> 0) then
                  AbortCLI ('invalid File Number')
                ELSE begin
                  { ----
                  IF (debug_488 > 4) then
                    writeln ('file num. is: ', file_num);
                  ---- }
                  CLOSE (data_file[file_num]);
                  file_rec[file_num].open := FALSE;
                end;
              end;
            end;
    end;
  Pop_Msg;
END;

{.pa}
```



```
Var _probe_ang : real;


Procedure init_cli_probe;
  begin
    enable_probe := false;
    _probe_ang   := 0.0;
  end;


Procedure cli_PROBE (token   : st1;              { sub command token }
                     Var buf : st128);           { command line      }
  Var   data   : st10;
        data1  : st1;
        r_data : real;
        v_code,
        err    : integer;
        inc_flg: boolean;

  BEGIN
   Push_Msg (' Proc: cli_PROBE');
    err := 0;
    IF (token = 'ANG') or (token = '') then begin
       IF ( FoundData(buf, data) ) then begin
          data1 := data;
          IF (data1 = 'INI') then begin
             _probe_ang := 0;
             PROBE ('init', _probe_ang);
          end
          ELSE begin
             IF (data[1] = 'I') then begin
                inc_flg := TRUE;
                DELETE (data, 1, 1);
             end
             ELSE
                inc_flg := FALSE;

             VAL (data, r_data, v_code);
             IF (v_code = 0) then begin
                IF (inc_flg) then
                   _probe_ang := _probe_ang + r_data
                ELSE
                   _probe_ang := r_data;
             end
             ELSE  err := 1;
          end;
       end
       ELSE  err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
  Pop_Msg;
 END; { cli_PROBE }
{.pa}
```

```
var _$rms  : record
               dev  : byte;
               cha  : byte;
               chb  : byte;
               tc   : real;
               mode : boolean;
               data : real;
             end;

Var _$mean : record
               dev  : byte;
               cha  : byte;
               tc   : real;
               data : real;
             end;


procedure init_cli_sae;
  begin
    with _$rms do begin
      dev := 0;  cha := 0;  chb := 0;  tc := 0.0;  data := 0.0;
    end;
    with _$mean do begin
      dev := 0;  cha := 0;  tc := 0.0;  data := 0.0;
    end;
  end;


Procedure cli_$RMS (token   : st3;              { sub command token  }
                    Var buf : st128);           { command line       }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

  BEGIN
   Push_Msg (' Proc: cli_$ae-RMS');
    err := 0;
    IF (token='CEA') or (token='CEB') or (token='TC') or (token='DIV')
       then begin
       IF ( FoundData( buf, data ) ) then begin
          VAL (data, r_data, v_code);
          IF (v_code = 0) then begin
             IF      (token = 'CEA') then _$rms.cha := TRUNC(r_data)
             ELSE IF (token = 'CEB') then _$rms.chb := TRUNC(r_data)
             ELSE IF (token = 'TC' ) then _$rms.tc  := r_data
             ELSE IF (token = 'DIV') then _$rms.dev := TRUNC(r_data)
          end
          ELSE  err := 1;
       end;
    end
    ELSE IF (token = 'MOD') then begin
       IF ( FoundData (buf, data) ) then begin
          data3 := data;
          IF (data3 = 'COS') then
             _$rms.mode := FALSE
          ELSE
             _$rms.mode := TRUE;
       end
       ELSE  err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
   Pop_Msg;
  END; { cli_$ae-RMS }
(.pa)

Procedure cli_$MEAN (token   : st3;             { sub command token }
                     Var buf : st128);          { command line      }
```



```
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

  BEGIN
   Push_Msg (' Proc: cli_$ae-MEAN');
     err := 0;
     IF ( token = 'CEA') or (token = 'TC' ) or (token = 'DIV') then begin
        IF ( FoundData( buf, data ) ) then begin
           VAL (data, r_data, v_code);
           IF (v_code = 0) then begin
              IF      (token = 'CEA') then _$mean.cha := TRUNC(r_data)
              ELSE IF (token = 'TC' ) then _$mean.tc  := r_data
              ELSE IF (token = 'DIV') then _$mean.dev := TRUNC(r_data)
           end
           ELSE  err := 1;
        end
        ELSE  err := 2;
     end
     ELSE  err := 3;
     DisplayError (err);
   Pop_Msg;
  END; { cli_$ae-MEAN }
(.pa)
```

```
var _$corr : record
               dev     : byte;
               chX     : byte;
               chY     : byte;
               Idelay  : real;
               Sdelay  : real;
               Tfactor : byte;
               Status  : byte;
               err_code: byte;
             end;

Var _$sam  : record
               dev     : byte;
               Trig    : byte;
               Proc    : byte;
               test    : byte;
               ACCTIM  : real;
               RealT   : real;
               RemainT : real;
               Result  : SAE_SAM_result_type;
               Status  : integer;
             end;


procedure init_cli_sam;
  var i : byte;
  begin
    with _$corr do begin
      dev := 0;     chX := 0;     chY := 0;
      Idelay := 0;  Sdelay := 0;  Tfactor := 0;  Status := 0;
    end;
    with _$sam do begin
      dev := 0;     Trig := 0;   Proc := 0;     test := 0;
      ACCTIM := 0;  RealT := 0;  RemainT := 0;  Status := 0;
      for i:=0 to 7 do
        Result[i] := 0;
    end;
  end;

{.pa}

Procedure cli_$corr (token   : st3;              { sub command token  }
                     Var buf : st128);           { command line       }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

  BEGIN
   Push_Msg (' Proc: cli_$ae-CORR');
     err := 0;
     IF    (token='CEX') or (token='CEY')
        or (token='IDE') or (token='SDE') or (token='TFA')
        or (token='DEV')
        then begin
        IF ( FoundData( buf, data ) ) then begin
           VAL (data, r_data, v_code);
           IF (v_code = 0) then begin
              IF      (token = 'CEX') then _$corr.chX     := TRUNC(r_data)
              ELSE IF (token = 'CEY') then _$corr.chY     := TRUNC(r_data)
              ELSE IF (token = 'IDE') then _$corr.Idelay  := r_data
              ELSE IF (token = 'SDE') then _$corr.Sdelay  := r_data
              ELSE IF (token = 'TFA') then _$corr.Tfactor := TRUNC(r_data)
              ELSE IF (token = 'DEV') then _$corr.dev     := TRUNC(r_data)
           end
           ELSE  err := 1;
        end;
    end
```



```
     ELSE  err := 3;
     DisplayError (err);
   Pop_Msg;
  END; { cli_$ae-CORR }

{.pa}

Procedure cli_$AM   (token   : st3;              { sub command token }
                     Var buf : st128);           { command line      }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err    : integer;

  BEGIN
   Push_Msg (' Proc: cli_$ae-SAM');
     err := 0;
     IF    (token = 'TRI') or (token = 'PRO' ) or (token = 'TES')
        or (token = 'ACC') or (token = 'DEV') then begin
        IF ( FoundData( buf, data ) ) then begin
           VAL (data, r_data, v_code);
           IF (v_code = 0) then begin
              IF      (token = 'TRI') then _$sam.Trig   := TRUNC(r_data)
              ELSE IF (token = 'PRO') then _$sam.Proc   := TRUNC(r_data)
              ELSE IF (token = 'TES') then _$sam.Test   := TRUNC(r_data)
              ELSE IF (token = 'ACC') then _$sam.ACCTIM := r_data
              ELSE if (token = 'DEV') then _$sam.dev    := TRUNC(r_data)
           end
           ELSE  err := 1;
        end
        ELSE  err := 2;
     end
     ELSE IF (token = 'STA') then begin
     end
     ELSE  err := 3;
     DisplayError (err);
   Pop_Msg;
  END; { cli_$ae-SAM }
{.pa}
```

```
Var _trav : array[0..3] of record
              posn  : real;
              tol   : real;
              wait  : byte;
              delay : byte;
              disp  : boolean;
            end;
    _trav_dev      : byte;
    _trav_offs     : real;
    _tra_cha       : byte;
    _tra_err       : boolean;


Procedure init_cli_trav;
  var i : byte;

  begin
    for i:=0 to 3 do
      with _trav[i] do begin
        posn := 0.0;  tol := 0.1;  wait := 100;  delay := 50;
      end;
    _trav_offs := 0.01667;                      { error correction constant }
    _trav_dev  := 0;
  end;


Procedure cli_TRAV (token   : st3;               { sub command token }
                    Var buf : st128);            { command line      }
  Var   data   : st10;
        data1  : st3;
        r_data : real;
        v_code,
        err, i : integer;
        inc_flg: boolean;

  Procedure Wait_Trav (com : st10);
    VAR   ctr, i              : integer;
          new_posn, posn, tol : real;
          old_posn            : array [0..2] of real;
          finish              : boolean;
    BEGIN
       Push_Msg (' Proc: Wait_Trav');
          finish := FALSE;
          _tra_err := FALSE;
          ctr := 0;
          new_posn := _trav[_tra_cha].posn;
          tol      := _trav[_tra_cha].tol;
          FOR i:=0 to 2 DO
             old_posn[i] := 0;
          REPEAT
             IF (upcase(com[1]) = 'A' ) then begin
                finish := TRUE;
                FOR i:=0 to 2 DO begin
                   TRAVER (read_d, i, new_posn);
                   _trav[i].posn := new_posn;
                   IF (new_posn <> old_posn[i]) then
                      finish := FALSE;
                   old_posn[i] := new_posn;
                end;
                UpdateWindow;
             end
             ELSE begin
                TRAVER (read_d, _tra_cha, posn);
                _trav[_tra_cha].posn := posn;
                IF (debug_488 > 5) then
                   writeln ('posn read d/w = ', posn:9:2);
                IF ((posn < new_posn + tol) and (posn > new_posn - tol)) then
                   finish := TRUE
                ELSE begin
                   UpdateWindow;
```



```
                   ctr := ctr + 1;
                end;
                IF ( (upcase(com[1]) = 'D') and (ctr > _trav[_tra_cha].delay) )
                   or
                   ( (upcase(com[1]) = 'W') and (ctr > _trav[_tra_cha].wait) )
                   then begin
                      write (bell);
                      IF (debug_488 > 5) then
                         writeln ('traver error in wait/delay');
                      _tra_err := TRUE;
                end;
             end;
          UNTIL (finish or _tra_err);
          UpdateWindow;

       Pop_Msg;
    END;


{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
BEGIN
 Push_Msg (' Proc: cli_TRAV');
   err := 0;

   IF ( token = 'X') or (token = 'Y') or (token = 'Z') or (token = 'R')
      or (token = '') or (token = 'DEV') or (token = 'DEL') then begin

      IF ( FoundData(buf, data) ) then begin
         data1 := data;

         IF (token[1] in ['X', 'Y', 'Z', 'R']) then begin
            if (token[1] = 'R') then
               _tra_cha := 3
            else
              _tra_cha := ABS ( INTEGER('X') - INTEGER(token[1]) );

            IF ( UPCASE (data[1]) = 'I') then begin
               inc_flg := TRUE;
               DELETE (data, 1, 1);
            end
            ELSE
               inc_flg := FALSE;

            VAL (data, r_data, v_code);

            IF (v_code = 0) then begin

               IF (inc_flg) then
                  _trav[_tra_cha].posn := _trav[_tra_cha].posn
                                          + r_data
               ELSE
                  _trav[_tra_cha].posn :=   r_data;

               r_data := _trav[_tra_cha].posn;

               IF (r_data > 0) then
                  r_data := r_data + _trav_offs
               ELSE
                  r_data := r_data - _trav_offs;

               DELAY (300);
               TRAVER (set_d, _tra_cha, r_data);
            end
            ELSE  err := 1;
         end

         ELSE IF (token = 'DEL') then begin

            IF (data[1] = 'W') then
               Wait_Trav ('wait')
```

```
            ELSE IF (data[1] = 'A') then
               Wait_Trav ('all finish')
            ELSE begin
               VAL (data, r_data, v_code);

               IF (v_code = 0) then begin
                  DELAY ( TRUNC(r_data * 1000.0));
                  Wait_Trav ('delay');
               end
               ELSE
                  err := 1;
            end;
         end

         ELSE IF (token = '') and (data3 = 'INI') then begin
            FOR i:=0 to 2 DO begin
               _trav[i].posn := 0.0;

               TRAVER (clear_d, i, _trav[i].posn);
            end;
            UpdateWindow;
         end

         ELSE IF (token = 'DEV') then begin
           Val (data, r_data, v_code);
           if (v_code = 0) then
             _trav_dev := trunc (r_data)
           else
             err := 1;
         end;
       end
       ELSE  err := 2;
    end
    ELSE  err := 3;
    DisplayError (err);
  Pop_Msg;
END; { cli_TRAV }
{.pa}
```



```
{-------------------------------------------------------------------}
{ DISA CTA Driver Routine                                           }
{ ----------------------                                            }
{  1) CTA_RMS  (com         :cta_dev_opt;{ I : read_d or set_d      }
{               dev_num     :byte;       { I : DISA 'DCBA device num.'}
{               Var chA     :byte;       {I/O: channel A #          }
{               Var tc      :real;       {I/O: time constant        }
{               Var OL_flag :boolean;    { 0 : True => overflow     }
{               Var data    :real);      {I/O: CTA RMS data         }

{  2) CTA_MEAN (com         :cta_dev_opt;{ I : read_d or set_d      }
{               dev_num     :byte;       { I : DISA 'DCBA device num.'}
{               Var chA     :byte;       {I/O: channel A #          }
{               Var tc      :real;       {I/O: time constant        }
{               Var it      :real;       {I/O: integration time     }
{               Var fil     :boolean;    {I/O: filter on/off        }
{               Var OL      :boolean;    {I/O: overflow flag        }
{               Var data    :real);      {I/O: MEAN value           }

{  3) SIG_COND (com         :cta_dev_opt;{ I : read_d or set_d      }
{               dev_num     :byte;       { I : DISA 'DCBA device num.'}
{               Var chA     :byte;       {I/O: channel A #          }
{               Var gain    :real;       {I/O: 1,2,5,10,.. 500      }
{               Var lp      :real;       {I/O: .1,.3,1,.. 300, OUT= -1}
{               Var hp      :real);      {I/O: .1,.3,1,.. 300, DC = -1}
{-------------------------------------------------------------------}

Type   cta_dev_opt      = dev_opt;

Const  cta_addr = $F80;

Var    cta_id   : integer;
       cta_err_msg  : st128;

Procedure Init_CTA_DRV;
   BEGIN
      WITH enable DO begin
         cta_rms   := FALSE;
         cta_mean  := FALSE;
         cond      := FALSE;
      end;
   END;
{.pa}

Procedure CTA_RMS (com         : cta_dev_opt;  { I : 'read_d' or 'set_d' }
                   dev_num     : byte;         { I : 'DCBA' dev. num.    }
                   Var chA     : byte;         {I/O: channel A #         }
                   Var tc      : real;         {I/O: time constant       }
                   Var OL_flag : boolean;      { 0 : 7 => overflow       }
                   Var data: real);            {I/O: CTA RMS data        }

  Const  cta_rms_addr = $818;

  Var    dum      : boolean;
         dev_addr : integer;
         data16   : array16;
         temp, i  : integer;

  BEGIN
    IF (not enable.cta_rms) then
       EXIT;

    Push_msg (' Proc: CTA RMS');
      dev_addr := cta_rms_addr or (dev_num shl 7);

      write_pc (cta_id, $30, cta_addr, 0);   { open CTA expansion bus }

      case com of
        read_d :
          begin
```

```
          write_pc (cta_id, $0B, dev_addr, 0);   { requst 16 word of data }
          if (read_pc (cta_id, 16, data16, cta_err_msg)) then
            printfln (bell + 'CTA RMS unit: ' + cta_err_msg);

          chA := data16[6] and $07 + 1;

          temp :=   (data16[2] and $0F) * 100
                  + (data16[1] and $0F) *  10
                  + (data16[0] and $0F);

          data := temp * 0.01;

          OL_flag := (data16[7] and $08) > 0;

          tc := 0.1;
          for i := 1 to (data16[7] and $03) do
            tc := tc * 10;
        end;
      set_d :
        begin
          If      (tc >= 100.0) then temp  := $03  {   oo to 100.0 }
          Else If (tc >=  10.0) then temp  := $02  { 99.9 to  10.0 }
          Else If (tc >=   1.0) then temp  := $01  {  9.9 to   1.0 }
          Else (if tc < 1.0     then) temp := $00;
          write_pc (cta_id, $30, dev_addr, temp);

          write_pc (cta_id, $30, (dev_addr + 1), ((chA - 1) and $07));
        end;
      Else begin
        Disp_msg_stack;
        printfln ('Invalid (CTA-RMS) command');
        printfln ('');
      end;
    end;

  Pop_msg;
END; { Proc: CTA_RMS }
{.pa}

Procedure CTA_MEAN (com     : cta_dev_opt;  { I : 'read_d' or 'set_d' }
                    dev_num : byte;         { I : 'DCBA' dev. num.    }
                    Var chA : byte;         {I/O: channel A #         }
                    Var it  : real;         {I/O: integration time    }
                    Var fil : boolean;      {I/O: filter on/off       }
                    Var OL  : boolean;      {I/O: overflow flag       }
                    Var data: real);        {I/O: MEAN value          }

  Const  cta_mean_addr = $80;

  Var    dum      : boolean;
         dev_addr : integer;
         data16   : array16;
         temp, i  : byte;

  BEGIN
    IF (not enable.cta_mean) then
       EXIT;

    Push_msg (' Proc: CTA_MEAN');
      dev_addr := cta_mean_addr or (dev_num shl 7);

      write_pc (cta_id, $30, cta_addr,  0);  { open CTA expansion bus }

      case com of
        read_d :
          begin
            write_pc (cta_id, $0B, dev_addr, 0);  { requst 16 word of data }
            if (read_pc  (cta_id, 16, data16, cta_err_msg)) then
              printfln (bell + 'CTA MEAN unit: ' + cta_err_msg);
```



```
            chA  := data16[0] and $0f;
            if      (chA = 0) then cha := 15
            else if (chA = 1) then cha := 16
            else                   cha := cha - 1;

            data :=   (data16[6] and $07) * 10.0
                    + (data16[5] and $0F) *  1.0
                    + (data16[4] and $0F) *  0.1
                    + (data16[3] and $0F) *  0.01
                    + (data16[2] and $0F) *  0.001
                    + (data16[1] and $0F) *  0.0001;

            if ((data16[7] and $08) = 0) then  data := data * -1.0;

            OL  := (data16[6] and $08) > 0;
            fil := (data16[7] and $01) > 0;

            it := 1.0;
            for i := 1 to ((data16[7] shr 1) and $03) do
              it := it * 10;
          end;

        set_d :
          begin
            If      (it >= 1000.0) then  temp := $06  { >= 1000.0 }
            Else If (it >=  100.0) then  temp := $04  { >=  100.0 }
            Else If (it >=   10.0) then  temp := $02  { >=   10.0 }
            Else (if it <    10.0  then) temp := $00;

            if fil then  temp := temp or $01;

            write_pc (cta_id, $30, dev_addr, temp);

            write_pc (cta_id, $30, (dev_addr + 7), (CEA + 1));
          end;

        Else begin
          printfln ('Invalid (CTA-MEAN) command');
          printfln ('');
        end;
      end;
    Pop_msg;
  END; { Proc: MEAN }
{.pa}

{ Signal Condition Driver Routines }
Procedure SIG_COND (com     : cta_dev_opt;  { I : 'read_d' or 'set_d      }
                    dev_num : byte;         { I : 'DCBA' dev. num.        }
                    Var chA : byte;         {I/O: channel A #             }
                    Var gain: real;         {I/O: 1,2,5,10, ...,500       }
                    Var lp  : real;         {I/O: .1,.3,1, .. 300, OUT= -1}
                    Var hp  : real);        {I/O: .1,.3,1, .. 300, DC = -1}

  Type r_array9 = array [0..8] of real;
       RorBtype = (real_d, binary_d);

  Procedure Decode_to (sub_com  : RorBtype;{ I : real_d else binary_d    }
                       Var b_code : byte;  {I/O: binary representation   }
                       Var r_code : real;  {I/O: real value GAIN, LP, HP }
                           spec   : r_array9);

    BEGIN
      Push_msg ( 'Sub Proc: DECODE_TO ');
        If (sub_com = real_d) then
          r_code := spec [b_code]

        Else begin
          b_code := 0;
          If (spec[8] > 0) then
            If (r_code >= spec[8]) then  b_code :=  8;
```

```
        If (b_code = 0) then begin
          If      (r_code >= spec[7]) then  b_code :=  7
          Else If (r_code >= spec[6]) then  b_code :=  6
          Else If (r_code >= spec[5]) then  b_code :=  5
          Else If (r_code >= spec[4]) then  b_code :=  4
          Else If (r_code >= spec[3]) then  b_code :=  3
          Else If (r_code >= spec[2]) then  b_code :=  2
          Else If (r_code >= spec[1]) then  b_code :=  1
          Else                              b_code :=  0;
        end;
      end;
    Pop_msg;
  END; { Sub_Proc: DECODE_TO }
{.pa}

  Const  cond_addr = $804;
         gain_spec : r_array9 = (1, 2, 5, 10, 20, 50, 100, 200, 500);
         lp_hp_spec: r_array9 = (0.1, 0.3, 1, 3, 10, 30, 100, 300, -1);

  Var    dum              : boolean;
         buf              : st128;
         data16           : array16;
         cond_ch_addr     : integer;
         temp1, i         : byte;

  BEGIN
    IF (not enable.cond) then
       EXIT;

    Push_msg (' Proc: SIG_COND');

      cond_ch_addr := cond_addr or (((dev_num + chA) and $0F) shl 7);

      write_pc (cta_id, $30, cta_addr,  0);     { open CTA expansion bus }

      case com of
        read_d :
          begin
            write_pc (cta_id, $08, cond_ch_addr, 0); { requst 16 word of data }
            if (read_pc  (cta_id, 16, data16, cta_err_msg)) then
              printfln (bell + 'SIGNAL COND. unit: ' + cta_err_msg);

            temp1 := data16[2] and $0F;                 { determining the GAIN   }
            decode_to (real_d, temp1, gain, gain_spec);
            temp1 := data16[1] and $0F;                 { determining the LP     }
            decode_to (real_d, temp1,    lp, lp_hp_spec);
            temp1 := data16[0] and $0F;                 { determining the HP     }
            decode_to (real_d, temp1,    hp, lp_hp_spec);
          end;

        set_d :
          begin
            write_pc (cta_id, $30, cta_addr,  0);  { open CTA expansion bus }
            decode_to (binary_d, temp1, gain, gain_spec);    { set GAIN to cta}
            write_pc (cta_id, $30, cond_ch_addr +1, temp1);
            decode_to (binary_d, temp1, lp,  lp_hp_spec);    { set LP to cta  }
            write_pc (cta_id, $30, cond_ch_addr +2, temp1);
            decode_to (binary_d, temp1, hp,  lp_hp_spec);    { set HP to cta  }
            write_pc (cta_id, $30, cond_ch_addr +3, temp1);
          end;

        Else begin
          printfln ('Invalid (CTA-Signal Cond.) command');
          printfln ('');
        end;
      end;
    Pop_msg;
  END; { Proc: SIG_COND }
{.pa}
```



```
{=============================================================}
type st10    = string[10];
     array16 = array[0..31] of byte;
VAR  debug_488   : byte;              {-depended on it's value, various    }
                                      { level of display will be generated }

{=============================================================}
{ Initialize IEEE 488 I/O Routine                             }
{ ----------------------------                                }
{  1) INIT_GPIB (gpib_name   : st40;     { I   : GPIB name in ibconf }
{                var gpib_id : integer   { O   : GPIB device ID #    }
{                var error   : st10)     { O   : error message       }
{               : boolean;               { ture if error occure      }
{-------------------------------------------------------------}

procedure ib_error (ib_name : st10; err_msg : st128);
   begin
     (* This routine would, among other things, check iberr to
        determine the exact cause of the error condition and then
        take action appropriate to the application.
        For errors during data transfers, ibcnt may be examined
        to determine the actual number of bytes transferred.
      *)
     err_msg := ib_name
                + ' ibsta=' + hex(ibsta)
                + ' iberr=' + hex(iberr)
                + ' ibcnt=' + hex(ibcnt);
   end;
type gpib_ntype = string[40];

function INIT_GPIB (gpib_name   : gpib_ntype; { I   : GPIB name in ibconf }
                    var gpib_id : integer;    { O   : GPIB device ID #    }
                    var error   : st128)      { O   : error message       }
                    : boolean;                { ture if error occure      }

BEGIN
  error := '';
  gpib_id := ibfind (gpib_name);
  if (gpib_id < 0) then begin
    init_gpib := true;
    error := 'ibfind (' + gpib_name + ')';
  end
  else begin
    ibclr (gpib_id);   ib_error ('ibclr', error);
    if (length(error) > 0) then
      init_gpib := true
    else
      init_gpib := false;
  end;
END; { Proc: INIT_GPIB }
{.pa}

{=============================================================}
{ ==== read data from National Instruments GPIB_PCIIA interface,}
{      at base I/O address $02E1                              }
{                                                             }
{      Convert the BUS_COMMAND, DEVICE_ADDRESS, & DATA information}
{      to a format which cna can transfer it to the IEEE-488 bus.}
{      This info. is received by the CTA IEEE-488 controller and}
{      thus converted to information the CTA can understand.  }
{                                                             }
{  1) Write_pc (GPIB_id    : Integer;     I = GPIB address     }
{               Bus_Com    : Integer;     I = $30 -> Write & set Remote}
{               Dev_Addr   : Integer;     I = Dev address (12 bits)}
{               Data       : Byte);       I = # bits data      }
{                                                             }
{ ==== read data form the DISA IEEE 488 card                  }
{  2) READ_PC (GPIB_id,                   { I GPIB address    }
{              Var num_of_byte: byte;     { I # of byte to read }
{              Var data   : array16;      { O data read from 488 }
```

```
{              err_msg    : st128)            { O error message      }
{              : boolean;                     { true if error occur  }
{                                                                    }
{====================================================================}

PROCEDURE WRITE_PC (GPIB_id, BUS_COM, DEV_ADDR :INTEGER;  DATA :BYTE );

VAR
   FORMAT_A, FORMAT_B :INTEGER;

BEGIN
  Push_msg (' Proc: Write pc');
    FORMAT_A := $C000
                + (BUS_COM AND $3F) SHL 8
                + (DEV_ADDR SHR 5);
    FORMAT_B := $8000
                OR (DEV_ADDR AND $0018) SHL 9
                OR (DEV_ADDR AND $0007) SHL 4
                OR (DATA AND $F0) SHL 4
                OR (DATA AND $0F);

    {**** SEND FORMAT B (lo, hi) then A (lo, hi) ****}
    ibbuf[1] := chr( lo( format_b));
    ibbuf[2] := chr( hi( format_b));
    ibbuf[3] := chr( lo( format_a));
    ibbuf[4] := chr( hi( format_a));

    ibwrt (gpib_id, ibbuf, 4);
  Pop_msg;
END; { Proc: WRITE_PC }
{.pa}

{ ==== read data from the DISA IEEE 488 card }
Function READ_PC (GPIB_id        : integer;        { I GPIB address       }
                  num_of_byte    : byte;           { I # of byte to read  }
                  Var data       : array16;        { O data read from 488 }
                  Var err_msg    : st128)          { O error message      }
                  : boolean;                       { true if error occur  }

  VAR max, addr, i, j            : integer;
      er_format, er_seq, er_valid : boolean;
      temp1, temp2               : byte;

  Function Validate_data : boolean;
    BEGIN
      max := (num_of_byte shr 1) - 1;          { max = (num_of_byte / 2)-1 }
      er_format := false;
      er_seq    := false;
      er_valid  := false;
      j := 1;

      For i:=0 to max do begin
        temp1 := integer (ibbuf[j]);
        temp2 := integer (ibbuf[j+1]);

        data[i] := (temp1 and $0F) or (temp2 shl 4);

        addr := ((temp2 shr 1) and $08) or ((temp1 shr 4) and $07);

        If ((addr and max) <> (max - i)) then     { check for addr error    }
          er_seq := true;

        If ((temp2 and $20) = 0) then             { check for data valid    }
           er_valid := false;  { <<-------------- this should be set to TRUE }

        If (i < max) then begin
          If ((temp2 and $C0) <> $80) then        { check for FORMAT B word }
            er_format := true;
        end
        Else begin
```



```
          If ((temp2 and $C0) <> $C0) then        { check for FORMAT A word }
            er_format := true;
        end;
        j := j + 2;
      end;

      Validate_data := not (er_format or er_seq or er_valid);
    END; { Func: Validate_data : boolean }

  var rsp : integer;

  BEGIN
    Push_msg (' Func: READ_PC ');
      err_msg := '';
      ibrsp (gpib_id, rsp);
      if ((rsp and $FF) <> $22) then begin
        Read_PC := true;
        err_msg := 'ibrsp=' + Hex(rsp);
        ibtrg (gpib_id);
        ibclr (gpib_id);
      end

      else begin
        ibrd (gpib_id, ibbuf, num_of_byte);

        if (not validate_data) then begin
          Read_PC := true;
          If er_format then
            err_msg := 'Format ';
          If er_seq then
            err_msg := err_msg + 'Sequence ';
          If er_valid then
            err_msg := err_msg + 'Invalid Data';
        end

        else begin
          Read_PC := false;
        end;
      end;
    Pop_msg;
  END; { Read_PC (..) : boolean }
{.pa}
```

```
{* TURBO Pascal Declarations                                    *)

{*$V-*)          (* relax string length restrictions           *)

Const
      MAXIBBUF = $100;  (* maximum buffer size for I/O functions         *)
      MAXINTBUF = $100; (* maximum buffer size for integer I/O functions *)

Type
       icbuf =  array[1..MAXIBBUF] of char;
       iclbuf = array[1..MAXINTBUF] of integer;
       ibstring =  string[50];
       str4 = string[4];
 Var
    ibsta : integer;   (* status word                 *)
    iberr : integer;   (* GPIB error code             *)
    ibcnt : integer;   (* number of bytes sent or, in the event of *)
                       (* DOS error, the DOS error code           *)
    ibbuf : icbuf;     (* I/O buffer for commands/data   *)
    intbuf : iclbuf;   (* integer I/O buffer for data    *)
    bdname : ibstring;(* board or device name           *)
    bd,dvm : integer;  (* Board descriptor               *)
    vcnt: integer;     (* v or byte count                *)
    flname : ibstring;(* file name                      *)
    mask : integer;    (* wait mask for IBWAIT ftn.      *)
    ppr,spr:integer;   (* parallel,serial poll responses *)


{* Ibfn is the common entry point into the language interface, tpib.com.   *)
{* Its arguments are generalized to meet the needs of each individual GPIP *)
{* function, and are decoded as follows:                                   *)
{*          ibfnasm (name,iberr,ibcnt,buf,ibuf,vcnt,bd,fcode,iberr,ibcnt)  *)
{* where:       name  = string for bdname, flname, bname                   *)
{*              iberr = GPIB-PC error code                                 *)
{*              ibcnt = GPIB-PC count                                      *)
{*              ibuf  = integer array for (var) rdi, wrti buffers          *)
{*              buf   = integer array for (var) rd, wrt, cmd buffers       *)
{*              vcnt  = integer for v, cnt, (var) spr, (var) ppr           *)
{*              bd    = integer for bd                                     *)
{*              fcode = integer for function code                          *)

function ibfn (name:ibstring;var iberr,ibcnt:integer;var buf:icbuf;var bufl:iclbuf;
      var vcnt:integer;bd,fncode:integer):integer; external 'dd-ni488.bin';


{* You MUST include the appropriate declaration, as            *)
{* given below, for each procedure or function you call.       *)
{* You may omit declarations for functions you do not call.    *)

procedure    ibbna (bd:integer;bname:ibstring);
    var
    name : ibstring;
    begin
    name := bname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,26);
    end;
procedure    ibcac (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,16);
    end;
procedure    ibclr (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,22);
    end;
procedure    ibcmd (bd:integer;ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,32);
```



```
    end;
procedure    ibcmda (bd:integer;ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,33);
    end;
procedure    ibdiag (bd:integer;ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,34);
    end;
procedure    ibdma (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,11);
    end;
procedure    ibeos (bd:integer;v:integer);
    begin
ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,12);
    end;
procedure    ibeot (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,14);
    end;
function     ibfind (bdname:ibstring):integer;
    var
    name : ibstring;
    begin
    name := bdname + chr(0);
    ibfind := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,27);
    end;
procedure    ibgts (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,15);
    end;
procedure    ibist (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,10);
    end;
procedure    ibloc (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,5);
    end;
procedure    ibonl (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,1);
    end;
procedure    ibpad (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,8);
    end;
procedure    ibpct (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,24);
    end;
procedure    ibppc (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,7);
    end;
procedure    ibrd (bd:integer;var ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,28);
    end;
procedure    ibrdi (bd:integer;var intbuf:iclbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,36);
    end;
procedure    ibrda (bd:integer;var ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,29);
    end;
procedure    ibrdf (bd:integer;flname:ibstring);
    var
```

```
    name : ibstring;
    begin
    name := fname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,17);
    end;
procedure    ibrpp (bd:integer;var ppr:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,ppr,bd,19);
    end;
procedure    ibrsc (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,2);
    end;
procedure    ibrsp (bd:integer;var spr:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,spr,bd,25);
    end;
procedure    ibrsv (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,6);
    end;
procedure    ibsad (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,9);
    end;
procedure    ibsic (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,3);
    end;
procedure    ibsre (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,4);
    end;
procedure    ibstop (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,21);
    end;
procedure    ibtmo (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,13);
    end;
procedure    ibtrg (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,23);
    end;
procedure    ibwait (bd:integer;mask:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,mask,bd,0);
    end;
procedure    ibwrt (bd:integer;ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,33);
    end;
procedure    ibwrti (bd:integer;intbuf:iolbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,37);
    end;
procedure    ibwrta (bd:integer;ibbuf:icbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,31);
    end;
procedure    ibwrtf (bd:integer;fname:ibstring);
    var
    name : ibstring;
    begin
    name := fname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,18);
    end;
```

```
{-------------------------------------------------------------------}
{ Serial to/from IEEE 488 I/O Routine                               }
{-------------------------------------------------------------------}

VAR   max_read_pc : byte;              {-max num. of attempt to resolve err }


{-------------------------------------------------------------------}
{ Serial to/from IEEE 488 I/O Routine                               }
{ ---------------------------------                                 }
{  1) Read_488  (Var Buf : st128;    Return the text from IOtech    }
{                Err_Msg : st128);   ERROR message                  }
{                                                                   }
{  2) Write_488 (Buf     : st128;    Send this text to IOtech SYS 488 }
{                Err_Msg : st128);   ERROR message                  }
{                                                                   }
{  3) Init_IOtech;                   Set COM1: Baud rate to 9600    }
{                                              No par., 8 bit, 1 stop }
{                                    Initial Serial chip            }
{                                    Reset & Initialize SYSTEM 488  }
{-------------------------------------------------------------------}

{ ==== read the IOtech 488 through the serial port }
Procedure READ_488 ( Var BUF : st128; ERR_MSG : st128);
  Var  i, in_stat, data, old_data  : integer;

  BEGIN
    Push_msg (' Proc: Read_488');
      i := 0;    buf := '';     old_data := 0;

      DELAY (25);

      Repeat
        Port [$3fc] := 3;
        in_stat := Port [$3fd];
        Port [$3fc] := 2;
        If ( (in_stat and 1) = 1 ) then begin
          old_data := data;
          data := Port [$3f8];
          buf := buf + chr (data);
        end;
        Delay (5);
        i := i + 1;
        If (i > 5000) then begin
          Push_msg ('  *ERROR* Occur in Reading --> ' + err_msg);
          Push_msg ('  ** buf =' + buf );
          Abort;
        end;
      Until ( (old_data = 13) and (data = 10) );
      If (debug_488 > 8) then  Writeln ('read 488 (end) buf = ', buf);
    Pop_msg;
  END; { Proc: READ_488 }
(.pa)

{ ==== write data through the serial port to IOtech 488 }
Procedure WRITE_488 ( BUF : st128; ERR_MSG : st128);
  Var  i,j  : integer;
       err  : st128;

  BEGIN
    Push_msg (' Proc: Write 488');

      Delay-(25);

      Port [$3fc] := 2;                          { Disable DTR }
      buf := 'S' + buf + ^M + ^J;

      For i := 1 to Length (buf) Do begin
        j := 0;
        While ((Port [$3fd] and $20) = 0) do begin
```



```
          j := j +1;
          If (j > 500) then begin
            Push_msg ('  ** ERROR, during the send loop -> ' + err_msg);
            Abort;
          end;
        end;
        Port [$3f8] := integer (buf[i]);
      end;
    Pop_msg;
  END; { Proc: WRITE_488 }
(.pa)

{ ==== Initialize the Iotech serial to IEEE488 bus controller to
       - 9600 buad, no parity, 1 stop bit and 8 bits data word.
       - this also init. the IEEE488 with the EOI terminator.          }
PROCEDURE INIT_IOTECH;

VAR  BUF    : st128;

BEGIN
  Port [$3fb] := $80;       { set line-control register for setting baud rate}
  Port [$3f8] := $0c;       { LSB of baud rate = 9600                        }
  Port [$3f9] := $00;       { MSB of baud rate                               }
  Port [$3fb] := $03;       { set Parity=none, Stop bit=1 & Char len=8        }
  PORT[$3F9] := 0;          { DISABLE ALL FOUR CLASSES OF INTERRUPTS         }
  PORT[$3FC] := 2;          { DISABLE DTR                                    }

  Write_488 ( '8', ' *Resetting System 488');
  DELAY (1500);             { the min. delay is 1500 ms **                   }
  BUF := 'TERM7;EOI';       { initialize IOcech not to use any terminator    }
  WRITE_488 ( BUF, 'Initializing IOtech');
  max_read_pc := 5;         { to resolve read 488 error                      }
END; { Proc: INIT_IOTECH }
(.pa)

{============= HOT-WIRE PROBE Rotation Driver ====================}
CONST probe_delay = 5;          { length of the low stepping pulse    }
      probe_step  = 7.5;        { min. angular stepping angle         }

VAR   old_angle : real;


{ - - HOT-WIRE PROBE Rotation Driver - - - - - - - - - - - - - - - - }

Procedure PROBE (com       : st10;
                 Var angle : real);
  VAR  temp : integer;
       i    : byte;
  BEGIN
     IF (not enable.probe) then
        EXIT;
     Push_msg (' Proc: PROBE');
       CASE  UPCASE(com[1])  of
          'I' : begin       { Initialize angle  }
                  angle := 0;
                  old_angle := 0;
                  Write_488 ('OU;C1X',' Initialize IOtech digital ports');
                end;
          'S' : begin       { set probe absolute angle }
                  temp := TRUNC((angle - old_angle) / probe_step);
                  IF (temp > 0) then
                    Write_488 ('OU;B1;1X', ' CW Rotation')
                  ELSE
                    Write_488 ('OU;B1;0X', ' CCW Rotation');
                  DELAY (3 * probe_delay);

                  FOR i:=1 to ABS (temp) DO begin
                    Write_488 ('OU;B2;1X', ' rising edge');
                    DELAY (probe_delay);
                    Write_488 ('OU;B2;0X', ' falling edge');
```

```
                    DELAY (probe_delay);
                    IF temp > 0 then begin
                       old_angle := old_angle + probe_step;
                       IF old_angle >  360 then old_angle := old_angle - 360;
                    end
                    ELSE begin
                       old_angle := old_angle - probe_step;
                       IF old_angle < -360 then old_angle := old_angle + 360;
                    end;
                  end;
                  angle := old_angle;
                end;
          'R' : begin       { reading probe angle }
                  angle := old_angle;
                end;
          ELSE  begin
                  disp_msg_stack;
                  writeln ('  ** invalid command >> ', com, ' <<');
                end;
        end;
      Pop_msg;
  END;

(* ---- Debug Code ---- *>
  procedure Init_IOtech;
  begin
  end;

  procedure probe (com :st10; var angle :real);
  begin
    angle := 0.0;
  end;
<* ---- Debug Code ---- *)
```



```
{-------------------------------------------------------------------}
{ DISA SAE Driver Routine                                           }
{ ----------------------                                           }
{  1) SAE_RMS  (com          :sae_dev_opt;{ I : read or set         }
{               dev_num      :byte        { I : 'DCBA' device num.  }
{               Var chA      :byte;       {I/O: channel A #         }
{               Var chB      :byte;       {I/O: channel B #         }
{               Var tc       :real;       {I/O: time constant       }
{               Var rms_mode:boolean;     { O : T =RMS, F =CORR     }
{               Var data     :real);      {I/O: RMS or CORR data    }
{                                                                   }
{  2) SAE_MEAN (com          :sae_dev_opt;{ I : read or set         }
{               dev_num      :byte        { I : 'DCBA' device num.  }
{               Var chA      :byte;       {I/O: channel A #         }
{               Var tc       :real;       {I/O: time constant       }
{               Var data     :real);      {I/O: MEAN value          }
{-------------------------------------------------------------------}

Type   sae_dev_opt      = dev_opt;

Const  sae_addr = $780;

Var    sae_id       : integer;
       sae_err_msg  : st128;

Procedure Init_SAE_DRV;
   BEGIN
      WITH enable DO begin
         sae_rms   := FALSE;
         sae_mean  := FALSE;
         sae_aam   := FALSE;
         sae_corr  := FALSE;
      end;
   END;
{.pa}

Procedure SAE_RMS (com          : sae_dev_opt;{ I : 'read_d' or 'set_d' }
                   dev_num      : byte;       { I : 'DCBA' dev. num.    }
                   Var chA      : byte;       {I/O: channel A #         }
                   Var chB      : byte;       {I/O: channel B #         }
                   Var tc       : real;       {I/O: time constant       }
                   Var rms_mode: boolean;     { O : T =RMS, F =CORR     }
                   Var data     : real);      {I/O: RMS or CORR data    }

  Const  sae_rms_addr = $048;

  Var    dum      : boolean;
         dev_addr : integer;
         data16   : array16;
         temp, i  : byte;

  BEGIN
    IF (not enable.sae_rms) then
       EXIT;

    Push_msg (' Proc: SAE RMS');
      dev_addr := sae_rms_addr or (dev_num shl 7);

      write_pc (sae_id, $30, sae_addr, 0);   { open SAE expansion bus }

      case com of
        read_d :
          begin
            write_pc (sae_id, $0B, dev_addr, 0);   { request 16 word of data }
            if (read_pc (sae_id, 16, data16, sae_err_msg)) then
              printfln (bell + 'SAE RMS unit: ' + sae_err_msg)
            else begin
              chA := (data16[6] and $0E) shr 1;
              chB := ((data16[7] and $0C) shr 2)
                     or ((data16[6] and $01) shl 2);
```

```
            rms_mode := ((data16[3] and $01) = $01);

            If (rms_mode) then begin
              data :=   (data16[2] and $0F) * 1.0
                      + (data16[1] and $0F) * 0.1
                      + (data16[0] and $0F) * 0.01;
            end
            Else begin
              data :=   (data16[2] and $0F) * 10.0
                      + (data16[1] and $0F) *  1.0
                      + (data16[0] and $0F) *  0.1;
            end;

            If ((data16[3] and $08) = $00) then
              data := data * -1.0;

            tc := 0.1;

            For i := 1 to (data16[7] and $03) do
              tc := tc * 10;
          end;
        end;

        set_d :
          begin
            temp := (chA shl 5);
            temp := temp or (chB shl 2);

            If      (tc >= 100.0) then temp := temp or $03  (   oo to 100.0 )
            Else If (tc >=  10.0) then temp := temp or $02  ( 99.9 to  10.0 )
            Else If (tc >=   1.0) then temp := temp or $01; (  9.9 to   1.0 )

            write_pc (sae_id, $30, dev_addr, temp);
          end;

        Else begin
          Disp_msg_stack;
          printfln ('Invalid (SAE-RMS) command');
          printfln ('');
        end;
      end;

    Pop_msg;
  END; { Proc: SAE RMS }
{.pa}

Procedure SAE_MEAN (com     : sae_dev_opt;  { I : read_d or set_d }
                    dev_num : byte;         { I : 'DCBA' dev. num.}
                    Var chA : byte;         {I/O: channel A #     }
                    Var tc  : real;         {I/O: time constant   }
                    Var data: real);        {I/O: MEAN value      }

  Const  sae_mean_addr = $050;

  Var    dum      : boolean;
         dev_addr : integer;
         data16   : array16;
         temp, i  : byte;

  BEGIN
    IF (not enable.sae_mean) then
       EXIT;

    Push_msg (' Proc: SAE MEAN');
      dev_addr := sae_mean_addr or (dev_num shl 7);

      write_pc (sae_id, $30, sae_addr, 0);   { open SAE expansion bus }

      Case com of
        read_d :
```



```
          begin
            write_pc (sae_id, $0B, dev_addr, 0); { request 16 word of data}
            if (read_pc  (sae_id, 16, data16, sae_err_msg)) then
              printfln (bell + 'SAE MEAN unit: '+ sae_err_msg)
            else begin
              chA  :=  (data16[6] and $0E) shr 1;
              data :=  (data16[3] and $0F) * 1.0
                     + (data16[2] and $0F) * 0.1
                     + (data16[1] and $0F) * 0.01
                     + (data16[0] and $0F) * 0.000625;
              tc := 1.048576;

              For i := 1 to (data16[7] and $03) do
                tc := tc * 10;
            end;
          end;

        set_d :
          begin
            temp := (chA shl 5);

            If      (tc >= 1048.576) then temp := temp or $03  ( >= 1048.576 )
            Else If (tc >= 104.8576) then temp := temp or $02  ( >= 104.8576 )
            Else If (tc >= 10.48576) then temp := temp or $01; ( >= 10.48576 )

            write_pc (sae_id, $30, dev_addr, temp);
          end;

        Else begin
          Disp_msg_stack;
          printfln ('Invalid (SAE-MEAN) command');
          printfln ('');
        end;
      end;
    Pop_msg;
  END; { Proc: SAE MEAN }
```

```
type Barray4 = array[0..3] of byte;

function IntReal (Var data : byte) : real;
  var Bdata : Barray4 ABSOLUTE data;
  begin
    IntReal  := (Bdata[3] * 16777216.0)
              +(bdata[2] * 65536.0)
              +(Bdata[1] * 256.0)
              + Bdata[0];
  end;

procedure RealInt ( x : real;  Var Bdata : Barray4 );
  var  t, r  : real;
       i     : byte;
  begin
    for i:=0 to 3 do begin
      t := (x / 256.0);
      r := frac (t);
      x := int  (t);
      Bdata[i] := trunc(r * 256.0);
    end;
  end;

type SAE_SAM_result_type = array [0..7] of real;

var  SAE_SAM_process_mode : byte;
     SAE_SAM_int_time     : real;

Procedure SAE_SAM  (com             : sae_dev_opt;      { I : read_d or set_d          }
                    dev_num         : byte;             { I : 'DCBA' dev. num.         }
                    Var Trigger_mode :byte;             {I/O: 0..3                     }
                    Var Process_mode :byte;             {I/O: 0..31                    }
                    Var HW_test_bit  :byte;             {I/O: 0..1                     }
                    Var ACCTIM       :real;             {I/O: 0.000250 .. 687194.75 sec.}
                    Var RemainTime   :real;             { O : real                     }
                    Var RealTime     :real;             { O : 0.000080 .. 343597.28 sec.}
                    Var Result       :SAE_SAM_result_type;{ O : real                    }
                    Var Status_code  :integer);         { O : integer                  }

  Const  sae_sam_addr = $060;
         Two32        = 4294967295.0;

  Var    dev_addr     : integer;
         data16       : array16;
         temp, i      : byte;
         Bdata        : Barray4;
         INTTIM       : real;
         flag         : boolean;
         timeOutCtr   : byte;
         DeltaT       : real;

  BEGIN
    IF (not enable.sae_sam) then
       EXIT;

    Push_msg (' Proc: SAE SAM');
      dev_addr := sae_sam_addr or (dev_num shl 7);

      write_pc (sae_id, $30, sae_addr, 0);   { open SAE expansion bus }

      Case com of
        read_d :
          begin
            { ------------------------------------------------------------- }
            { clear the result to 0                                         }
            for i:= 0 to 7 do
              result[i] := 0.0;

            { signal the SAM unit to update its registers   >> UP=1,F=0,A=0 }
            temp :=   $80 or (HW_test_bit  and $01);
```



```
            write_pc  (sae_id, $30, (dev_addr + 1), temp);

            { wait for update to complete                   >> DU = 1       }
            TimeOutCtr := 0;

            repeat
              TimeOutCtr := TimeOutCtr + 1;

              { request 1 word of information from SAM                      }
              write_pc (sae_id, $09, dev_addr, 0);
              flag := read_pc (sae_id, 4, data16, sae_err_msg);
            until (  flag
                  or ((data16[0] and $80) = $80)
                  or (TimeOutCtr > 11)          );

            if flag then begin
              printfln (bell + 'SAM READ err: ' + sae_err_msg);
              exit;
            end;

            if (TimeOutCtr > 11) then begin
              printfln (bell + 'SAM READ err: TIME OUT');
              exit;
            end;

            flag := false;

            { Request and Received FORMAT 0 data from the SAM unit          }
            temp := $1E or (HW_test_bit and 1);     { Setup SAM read format }
            write_pc  (sae_id, $30, (dev_addr + 1), temp);

            write_pc (sae_id, $0C, dev_addr, 0);   {request 32 word of data }
            if (read_pc  (sae_id, 32, data16, sae_err_msg)) then begin
              printfln (bell + 'SAM F0 : '+ sae_err_msg);
              flag := true;
            end
            else begin
              Realtime := (data16[13] * 256.0) + data16[12];

              INTTIM := IntReal(data16[8]);

              Result[0] := IntReal(data16[4]);
              Result[1] := IntReal(data16[0]);
            end;

            { ------------------------------------------------------------- }
            { Request and Received FORMAT 1 data from the SAM unit          }
            temp := $3E or (HW_test_bit and 1);     { Setup SAM read format }
            write_pc  (sae_id, $30, (dev_addr + 1), temp);

            write_pc (sae_id, $0C, dev_addr, 0);   {request 32 word of data }
            if (read_pc  (sae_id, 32, data16, sae_err_msg)) then begin
              printfln (bell + 'SAM F1 : '+ sae_err_msg);
              flag := true;
            end
            else begin
              Realtime :=  (data16[13] * 16777216.0)
                         + (data16[12] * 65536.0)
                         + Realtime;

              Realtime  := Realtime * 80.0e-6;

              Result[2] := IntReal(data16[8]);
              Result[3] := IntReal(data16[4]);
              Result[4] := IntReal(data16[0]);
            end;

            { ------------------------------------------------------------- }
            { Request and Received FORMAT 2 data from the SAM unit          }
            temp := $5E or (HW_test_bit and 1);     { Setup SAM read format }
```

```
        write_pc  (sae_id, $30, (dev_addr + 1), temp);

        write_pc (sae_id, $0C, dev_addr, 0);   {request 32 word of data }
        if (read_pc  (sae_id, 32, data16, sae_err_msg)) then begin
          printfln (bell + 'SAM F2 : '+ sae_err_msg);
          flag := true;
        end
        else begin
          Status_code := (data16[15] shl 8) or data16[14];
          Process_mode:= (data16[15] and $1F);
          HW_test_bit := (data16[14] and $01);

          Result[5] := IntReal(data16[8]);
          Result[6] := IntReal(data16[4]);
          Result[7] := IntReal(data16[0]);
        end;

        if (not flag) then begin
          if (((data16[14] shr 1) and $03) = $03) then
            printfln (bell + 'SAM READ err: Integration not stated')

          else begin
            RemainTime := 0.0;

            if ( (INTTIM > 0.5) and (((data16[14] shr 2) and $01) = 0) ) then
              INTTIM := INTTIM + 0.5;

            if ( INTTIM > 0.5 ) then
              RemainTime := 0.000016 * (Two32 + 2 - INTTIM);

            if ( INTTIM < 0.5 ) then
              INTTIM := Two32;

            DeltaT :=  (SAE_SAM_int_time * 1e6)
                     - ((Two32 - INTTIM -0.5) * 160 );

            ACCTIM := DeltaT * 1e-6;

            if (process_mode <> SAE_SAM_process_mode) then
              printfln (bell + 'SAM READ err: Unexpected processing mode')
            else begin
              for i:=0 to 7 do begin
                if (Result[i] = (Two32 - 1)) then
                  Result[i] := 0.0
                else begin
                  case process_mode of
                    17,21
                        : Result[i] :=  100.0
                                      * ( (Result[i]/deltaT) - 0.5 );
                    0,1,8..15
                        : Result[i] :=  200.0
                                      * (  (Result[i] - (deltaT/2.0) )
                                        / deltaT );
                    2
                        : Result[i] :=  20.0
                                      * (  (Result[i] - (deltaT/2.0) )
                                        / deltaT );
                    16
                        : Result[i] :=  10.24 * (Result[i] / deltaT);

                    17,24,26,27,28,30,31
                        : Result[i] :=  1.0 * (Result[i] / deltaT);
                  end;
                end;
              end;
            end;
          end;
        end;
      end;
```



```
      set_d :
        begin
          { recover form possible reset              >> T1,T0 = (0,1)    }
          write_pc (sae_id, $30, dev_addr, $40);

          { request SAM to update its registers      >> UP = 1           }
          temp :=   $80 or (HW_test_bit  and $01);
          write_pc  (sae_id, $30, (dev_addr + 1), temp);

          { wait for update to complete              >> DU = 1           }
          repeat
            { request 1 word of information from SAM                     }
            write_pc (sae_id, $09, dev_addr, 0);
            flag := read_pc (sae_id, 4, data16, sae_err_msg);
          until (  flag
                or ((data16[0] and $80) = $80) );

          if flag then
            printfln (bell + 'SAM SET err: ' + sae_err_msg)

          else begin
            SAE_SAM_process_mode := process_mode;
            SAE_SAM_int_time     := ACCTIM;

            { Send a stop command to the SAM unit    >> reg(0) = 0       }
            write_pc  (sae_id, $30, dev_addr, $00);

            { Select mode                            >> reg(0) = mode    }
            temp :=   ( Process_mode and $1F);
            write_pc  (sae_id, $30, dev_addr, temp);

            { Set up register (1)                    >> UP=0, F=0, A=0   }
            temp :=   (HW_test_bit  and $01);
            write_pc  (sae_id, $30, (dev_addr + 1), temp);

            { Convert the INTEGRATION TIME (real) to a 32 bit integer    }
            INTTIM  := Two32 + 2 - int( (ACCTIM / 0.000160) + 0.5 );
            RealInt (INTTIM, Bdata);

            write_pc  (sae_id, $30, (dev_addr + 4), Bdata[3]);
            write_pc  (sae_id, $30, (dev_addr + 5), Bdata[2]);
            write_pc  (sae_id, $30, (dev_addr + 6), Bdata[1]);
            write_pc  (sae_id, $30, (dev_addr + 7), Bdata[0]);

            { Set process mode & set trigger command                     }
            temp :=   ((Trigger_mode and $03) shl 6)
                   or ( Process_mode and $1F);
            write_pc  (sae_id, $30, dev_addr, temp);
          end;
        end;

      Else begin
        Disp_msg_stack;
        printfln ('Invalid (SAE SAM) command');
        printfln ('');
      end;
    end;
  Pop_msg;
END; { Proc: SAE SAM }
{.pa}

Procedure SAE_CORR (con          :sae_dev_cpt;  { I : read_d or set_d }
                    dev_num      :byte;         { I : 'DCBA' dev. num.}
                    Var chA      :byte;         {I/O: Channel A #     }
                    Var chB      :byte;         {I/O: Channel B #     }
                    Var Idelay   :real;         {I/O: Initial delay   }
                    Var Sdelay   :real;         {I/O: Step delay      }
                    Var Tfactor  :byte;         {I/O: Time factor     }
                    Var Status   :byte;         {I/O: Status/Mode byte}
                    Var err_code :byte);        {I/O: Error Code      }
```

```
Const  sae_corr_addr = $058;

Var    dum      : boolean;
       dev_addr : integer;
       data16   : array16;
       temp, i  : byte;
       Bdata    : Barray4;

BEGIN
  IF (not enable.sae_corr) then
     EXIT;

  Push_msg (' Proc: SAE_CORR');
    dev_addr := sae_corr_addr or (dev_num shl 7);

    write_pc (sae_id, $30, sae_addr, 0);   { open SAE expansion bus }

    Case com of
      read_d :
        begin
          write_pc (sae_id, $0B, dev_addr, 0); { request 16 word of data}
          if (read_pc  (sae_id, 16, data16, sae_err_msg)) then
            printfln (bell + 'SAE-CORR unit: '+ sae_err_msg)
          else
            Status := data16[0];
        end;

      set_d :
        begin
          temp :=   ( Status   and $C0)
                 or ((ChA      and $07) shl 3)
                 or ( ChB      and $07);
          write_pc (sae_id, $30, dev_addr, temp);

          write_pc (sae_id, $30, (dev_addr + 1), (Tfactor - 1));

          Idelay := Abs(Idelay);
          if (Idelay > 0.065537) then Idelay := 0.065537;
          RealInt (((Idelay / 1e-6) - 1), BData);
          write_pc (sae_id, $30, (dev_addr + 2), BData[1]);
          write_pc (sae_id, $30, (dev_addr + 3), BData[0]);

          Sdelay := Abs(Sdelay);
          if (Sdelay > 0.016384) then Sdelay := 0.016384;
          RealInt (((Sdelay / 1e-6) - 1), BData);
          write_pc (sae_id, $30, (dev_addr + 4), BData[1]);
          write_pc (sae_id, $30, (dev_addr + 5), BData[0]);

          write_pc (sae_id, $30, (dev_addr + 6), 0);
        end;

      Else begin
        Disp_msg_stack;
        printfln ('Invalid (SAE CORR) command');
        printfln ('');
      end;
    end;
  Pop_msg;
END; { Proc: SAE CORR }
{.pa}
```



```
{-------------------------------------------------------------------}
{ DISA TRAVER Driver Routine                                        }
{ ------------------------                                         }
{  1) Init_Traver                          { Reset calibration constant  }
{     TRAVER  (com          :trav_dev_opt;{ I : read, set or clear      }
{              channel_num :byte;         { I : channel num <0..2>      }
{              Var position:real);        {I/O: real in mm              }
{-------------------------------------------------------------------}

Type   trav_dev_opt  = dev_opt;

Var    trav_id       : integer;
       trav_err_msg  : st128;

TYPE Array3          = array [0..2] of byte;

VAR  pulse_per_mm    : array [0..3] of real;


Procedure Init_TRAV_DRV;
   BEGIN
      enable.trav  := FALSE;

      pulse_per_mm[0] := 240.0;
      pulse_per_mm[1] := 240.0;
      pulse_per_mm[2] := 240.0;
      pulse_per_mm[3] := 240.0;
  END;
{.pa}

{------------ TRAVER Driver ------------------------------------}
type trav_type = (data_c, position_c);

Procedure CONVERT_TO (com : trav_type;      { I : data or position       }
                      ch_num : byte;        { I : channel number         }
                      Var data: array3;     {I/O: 2's comp of pos. 24 bit}
                      Var position :real); {I/O: real in mm              }

  Var data_r         : array [0..2] of real;
      posn           : real;
      i, sign_code   : byte;

  BEGIN
    Push_msg (' Proc: CONVERT_TO');
      If (com = data_c) then begin       { convert to binary DATA }
        posn := int (position * pulse_per_mm[ch_num]);

        If (posn < 0.0) then begin
          sign_code := $FF;
          posn := abs( posn) -1;
        end
        Else
          sign_code := $00;

        If ((posn < 8388608.0) and (posn  >= 0)) then begin
          data_r[0] := posn / 65536.0;              { MSB    [0] }
          data_r[1] := frac( data_r[0] ) * 256.0;   { MSB-1   [1] }
          data_r[2] := frac( data_r[1] ) * 256.0;   { LSB    [2] }
          For i:=0 to 2 do begin
            data[i] := Trunc( data_r[i] ) xor sign_code;
          end;
        end
        Else begin
          Push_msg ('  ** invalid POSITION entry');
          Abort;
        end;
      end
      Else begin
        If (com = position_c) then begin     { convert to POSITION }
          If ((data[0] and $80) >0) then begin
```

```
          position := -1; sign_code := $FF;
        end
        Else begin
          position :=  1; sign_code := $00;
        end;
        position := position * (    ( data[0] xor sign_code ) * 65536.0
                                 + ( data[1] xor sign_code ) * 256.0
                                 + ( data[2] xor sign_code ) * 1.0 );
        position := position / pulse_per_mm[ch_num];
      end
      Else begin
        Push_msg ('  ** invalid COMMAND **');
        Abort;
      end;
    end;
  pop_msg;
end; { Proc: CONVERT_TO }
{.pa}

procedure TRAVER  (com         :trav_dev_cpt; { I : read_d, set_d or clear_d}
                   channel_num :byte;         { I : channel num <0..2>       }
                   Var position:real);        {I/O: real in mm               }

  Const ch_base  = $B60;
        no_com   = $00;
        clear_com= $10;    load_com = $20;
        run_com  = $40;    edir_com = $40;

  Var   data     : array3;
        data16   : array16;
        ch_num, i: integer;
        dum      : boolean;

  BEGIN
    IF (not Enable.trav) then
       EXIT;
    Push_msg (' Proc: TRAVER');
      ch_num := ch_base + ((channel_num and $03) shl 7);
      If (com = read_d) then begin       { load data to out reg. }
        write_pc (trav_id, $30, ch_num+6, load_com + run_com);
        write_pc (trav_id, $08, ch_num+6, no_com);   { requst 16 byte of data}
        If (read_pc  (trav_id, 16, data16, trav_err_msg)) then
          printfln (bell + 'TRAV unit: ' + trav_err_msg);
        data[0] := data16[2];
        data[1] := data16[4];
        data[2] := data16[6];
        convert_to (position_c, channel_num, data, position);
      end
      Else begin
        If (com = set_d) then begin
          convert_to (data_c, channel_num, data, position);
          write_pc (trav_id, $30, ch_num+0, data[2]);
          write_pc (trav_id, $30, ch_num+2, data[1]);
          write_pc (trav_id, $30, ch_num+4, data[0]);
          write_pc (trav_id, $30, ch_num+6, run_com);
        end
        Else begin
          If (com = clear_d) then begin
            write_pc (trav_id, $30, ch_num+6, clear_com); { clear counter }
            write_pc (trav_id, $30, ch_num+0, 0);         { set coord. to }
            write_pc (trav_id, $30, ch_num+2, 0);         { zero          }
            write_pc (trav_id, $30, ch_num+4, 0);
            write_pc (trav_id, $30, ch_num+6, run_com);
          end
          Else begin
            disp_msg_stack;
            printfln ('Invalid (TRAVER) command');
            printf ('');
          end;
        end;
```



```
      end;
    Pop_msg;
  END; { Proc: TRAVER }
{.pa}
```

```
{=============================================================}
{ Error Stack Routine :                                       }
{-------------------------------------------------------------}

Type    st40    = string[40];
        st128   = string[128];
        msg_ptr = ^msg_rec;
        msg_rec = record
                    msg  : st40;
                    next : msg_ptr;
                    prev : msg_ptr;
                  end;

Var     heaptop  : ^integer;
        first_msg, last_msg, new_msg : msg_ptr;

{=============================================================}
{ Convert an Integer to a Hex string                          }
{=============================================================}
type hex_type = string[10];

function HEX( DATA : INTEGER): hex_type;
  CONST
    HEX_STR       : ARRAY [0..15] Of CHAR
                    = ('0','1','2','3','4','5','6','7',
                       '8','9','A','B','C','D','E','F');
  BEGIN
    HEX := '$' + HEX_STR[ (DATA AND $F000) SHR 12 ]
               + HEX_STR[ (DATA AND $0F00) SHR  8 ]
               + HEX_STR[ (DATA AND $00F0) SHR  4 ]
               + HEX_STR[ (DATA AND $000F)        ];
  END; { Proc: HEX }
{=============================================================}
{.pa}

{=============================================================}
{ Error Stack Routine :                                       }
{ ---------------------                                       }
{   1)  Init_Msg_Stack;          * Get pointer to Heap        }
{                                                             }
{   2)  Push_Msg (Err_Msg);      * Push to message stack      }
{                                                             }
{   3)  Pop_Msg;                 * Remove message from stack  }
{                                                             }
{   4)  Disp_Msg_Stack;          * Non-distractive disp of stack }
{                                                             }
{   5)  Abort;                   * Display msg stack & Halt prog }
{-------------------------------------------------------------}

{ ==== initialize the error message stack }
Procedure INIT_MSG_STACK;
  BEGIN
    first_msg := nil;
    last_msg  := nil;
  END; { Proc; INIT_MSG_STACK }


{ ==== push error message to a dynamic stack }
Procedure PUSH_MSG ( ERR_MSG : st40 );
  Var  temp_ptr : msg_ptr;

  BEGIN
    new (new_msg);
    new_msg^.msg := err_msg;
    If (first_msg = nil) then begin
      first_msg := new_msg;
      temp_ptr  := nil;
    end
    Else begin
      last_msg^.next := new_msg;
```



```
      temp_ptr  := last_msg;
    end;
    last_msg := new_msg;
    last_msg^.next := nil;
    last_msg^.prev := temp_ptr;
  END; { Proc: PUSH_MSG }
{.pa}

{ ==== pop the error message when the procedure exec. correctly }
Procedure POP_MSG;
  Var temp_ptr : msg_ptr;

  BEGIN
    If (last_msg = nil) then begin
      Writeln ('  ** Unmatched POP_MSG error occur **');
      Halt;
    end
    Else begin
      If (last_msg = first_msg) then begin
        DISPOSE (last_msg);
        first_msg := nil;
        last_msg := nil;
      end
      Else begin
        temp_ptr := last_msg^.prev;
        temp_ptr^.next := nil;
        temp_ptr := last_msg;
        last_msg := last_msg^.prev;
        DISPOSE (temp_ptr);
      end;
    end;
  END; { Proc: POP_MSG }

{ ==== non-distructive display of the error message stack }
Procedure DISP_MSG_STACK;
  Var  Temp_ptr : Msg_ptr;

  BEGIN
    Writeln;
    Writeln ('Non-Districtive Display Of Message Stack');
    Writeln ('----------------------------------------');
    temp_ptr := first_msg;
    While (temp_ptr <> nil) do begin
      writeln (temp_ptr^.msg);
      temp_ptr := temp_ptr^.next;
    end;
  END; { Proc: DISP_MSG_STACK }
{.pa}

{ ==== ABORT the program and display the error message }
Procedure ABORT;
  BEGIN
    Writeln ;
    Writeln ('          ** Error Condition Occur **');
    Writeln ('A list of error messages from the most top level down');
    Writeln ('-----------------------------------------------');
    While (first_msg <> nil) do begin
      Writeln (first_msg^.msg);
      first_msg := first_msg^.next;
    end;
    Halt;
  END; { Proc: ABORT }
{.pa}
```

```
{*}
{*}
{*Source code copyright (c) 1986, by TurboPower Software*}
{*}
{*}

{.E-}

PROCEDURE FarIncoming;
 {-pick up control from a far call and transfer to near procedure}

CONST
 {following string must be here to find address from Main Code}
 idstring:BigTurboString='FARCALLHANDLER FOLLOWS';

BEGIN

 INLINE(

  {determine the address of the near procedure}
  $D1/$E0/{SHL    AX,1 - convert procnum to offset into
           integer array}
  $8B/$F0/{MOV    SI,AX - store in index register}
  $BB/ProcOffsets/{MOV    BX,offset(procoffsets)}
  $2E/$8B/$08/{MOV    CX,CS:[BX+SI] - offset to near call}

  {remove the bp,sp that were pushed on entry to this routine}
  $81/$C4/$04/$00/{ADD    SP,0004}

  {call the near procedure}
  $FF/$D1/{CALL    CX}

  {get far return address back from our local stack}
  $8B/$36/LocalStackPtr/{MOV    SI,localstackptr}
  $8B/LocalStack/{MOV    BX,offset(localstack)}
  $FF/$30/{PUSH    [BX+SI] - return segment on turbo stack}
  $FF/$70/$02/{PUSH    [BX+SI+02]}
  $8B/$6E/$04/{MOV    BP,[BX+SI+04]}
  $9C/{PUSHF - save flags}
  $81/$C6/$06/$00/{ADD    SI,0006 - regain stack space}
  $89/$36/LocalStackPtr/{MOV    localstackptr,SI}
  $9D/{POPF - restore flags}
  $CB{RETF    - back to calling scope}
  );
END;{FarIncoming}

{.E+}
```



```
procedure clean_input_cmd (var buf : st128);
  var i      : byte;
      token3 : st3;
  begin
    i := pos (';', buf);
    DELETE (buf, i+1, LENGTH(buf)-i);           { delete everything after ';'}

    if (i>1) then begin
      while (buf[1] = ' ') do delete (buf, 1, 1); { delete leading blanks    }

      FOR i:=1 to 3 DO begin                      { convert text to upcase   }
        buf[i]    := UPCASE (buf[i]);
      end;
      token3 := buf;

      if (token3 <> 'DOS') then begin
        DeleteCHAR ('+', buf);                    { remove all '+' & ' ' char}
        DeleteChar (' ', buf);
        FOR i:=1 to LENGTH(buf) DO                { convert text to upcase   }
          buf[i] := UPCASE (buf[i]);
      end;
    end;
  end; { clean_input_cmd }

{.pa}

Procedure Get_Input (msg : st10;
                     Var buf : st128);
  VAR  x, y, i             : integer;
       key_press           : boolean;
       temp_buf            : st128;
      start_str, cont_str : st80;
       token3              : st3;
       ch                  : char;

  procedure store_input (cont_prompt : st80);
    begin
      buf := buf + temp_buf;
      if ( length(buf) > 125 ) then begin
        buf := ';';
        printfln ('');
        printfln (bell + '...Line too long, text ignored...');
      end
      else begin
        i := pos (';', buf);
        if (i = 0) then begin
          if (temp_buf = '') then begin
            buf := buf + ';';
            i := pos (';', buf);
            printf (';');
          end
          else begin
            printfln ('');
            printf   (cont_prompt);
          end;
        end;
      end;
    end;

  BEGIN
     buf := '';
     IF (cmd_num >= 0) then begin
        IF (EOF(cmd_file[cmd_num])) then begin
           CLOSE (cmd_file[cmd_num]);
           cmd_file_exist[cmd_num] := FALSE;
           IF (cmd_num >= 0) then
              cmd_num := cmd_num - 1;
        end
        ELSE begin
          Str (cmd_num:1, start_str);
```

```
        start_str := start_str + msg;
        start_str := 'FILE:' + start_str;
        cont_str  := start_str + '# ';
        start_str :  start_str + ': ';
        println (''); printf (start_str);
        repeat
          READLN (cmd_file[cmd_num], temp_buf);
          printf (temp_buf);
          delay  (200);
          Store_input (cont_str);
        until (i>0);
      end
    end

    ELSE begin
       println ('');
       printf    ('CMD' + msg + '> ');
       TextColor      (W[input].foreground + blink);
       TextBackGround(W[input].background); printf(#16);
       x := WhereX;  y := WHEREY;
       key_press := false;
       REPEAT
          Update_Time_Date (false);
       UNTIL (keypressed);
       Read (KBD, ch);
       if (ch = ^[) then begin
         Read (KBD, ch);
         if (ch = #59) then            {F1}
           buf := 'HELP;'
         else if (ch = #60) then       {F2}
           buf := 'WINDOW;';
         Open_CMD_window;
         GOTOXY (x-1, y);
         printf (buf);
       end
       else if (ch = #13) then begin  (cr)
         buf := ';';
         Open_CMD_window;
         GOTOXY (x-1, y);
         printf (buf);
       end
       else begin
         Open_CMD_window;
         GOTOXY (x-1, y);
         printf (ch);
         buf := ch;
         Repeat
             READ (temp_buf);
             Enable_cursor;
             Store_input ( msg + '**     ');
         until (i>0);
       end;
    end;
    clean_input_cmd (buf);
END; { Get_Input }
```



```
{ - - Frequently used procedures & functions - - - - - - - - - - - - }

Procedure DeleteCHAR (ch : char;
                      Var Buf : st128); {I/O= contain string t.b.c.  }
  Var  i    : byte;
  BEGIN
     REPEAT
        i := POS (ch, buf);
        IF (i > 0) THEN
           DELETE (buf, i, 1);
     UNTIL (i=0);
  END;
{.pa}

Function  FoundCom (VAR buf     : st128; {I/O: line buffer            }
                    delim       : char;  { I : delimeter              }
                    Var com_buf : st10)  { O : return command string  }
                    : BOOLEAN;           { Return TRUE = command found}

  VAR i      : BYTE;
      found : BOOLEAN;
  BEGIN
     com_buf := '';
     i := POS (delim, buf);
     IF (i > 0) THEN BEGIN
        com_buf := COPY (buf, 1, i-1);
        DELETE (buf, 1, i);
        found := TRUE;
     END
     ELSE
        found := FALSE;
     FoundCom := found;
  END; { FoundCom }
{.pa}

Function FoundData (VAR buf      : st128; {I/O: line buffer            }
                    Var data_buf : st10)  { O : return data string     }
                    : BOOLEAN;            { Return TRUE if string found}

  Var found : BOOLEAN;
  BEGIN
     found := FoundCom (buf, ',', data_buf);
     IF (NOT found) THEN
       (found := FoundCom (buf, ';', data_buf);
     FoundData := found;
  END; { FoundData }
{.pa}

{ -- CURSOR related routines -- }
procedure SetCrtMode (CrtMode : integer);
  begin
    regs.ax := $00FF and CrtMode;
    Intr($10, regs);
  end;

procedure SetCurType (start_line, end_line : byte);
  begin
    regs.ah := $01;
    regs.ch := start_line;
    regs.cl := end_line;
    Intr($10, regs);
  end;

procedure Enable_cursor;
  begin
    SetCurType     (6, 7);                    { enable cursor underscore }
  end;

{* ------------------------------------------------------------------- **
procedure Hide_cursor;
```

```
  begin
    SetCurType     ($20, 15);                    { hide cursor }
  end;

Function GetVideoMode : integer;
  begin
    Intr($11, regs);
    GetVideoMode := regs.ax;
  end;

procedure GetCursor (Page                  : byte;
                     Var Row, Col, CurMode : byte);
  begin
    with regs do begin
      ax := $0300;
      bx := Page;
      Intr($10, regs);
      Row     := dh;
      Col     := dl;
      CurMode := cx;
    end;
  end;

procedure SetCursor (Page, Row, Col : byte);
  begin
    with regs do begin
      ah := 2;
      bh := Page;
      dh := Row;
      dl := Col;
      Intr($10, regs);
    end;
  end;

type SendStrType = string[128];

procedure SendStr (charStr : SendStrType);
  var i : byte;
  begin
    with regs do begin
      ah := $06;
      for i:=1 to length(charStr) do begin
        dl := integer(charStr[i]);
        msdos (regs);
      end;
    end;
  end;
{ ---------------------------------------------------------------------- }

{ -- TIME and DATE routines -- }
Type time_date_string = string[8];

function Get_time : time_date_string;
  var  time_str  : time_date_string;
       i         : byte;
       buf2      : string[2];

  BEGIN
     WITH regs DO begin
        ax := $2C00;
        MSDOS (regs);
        STR (HI(cx):2, buf2);   time_str := buf2 + ':';           { hour }
        STR (LO(cx):2, buf2);   time_str := time_str + buf2 + ':'; { min  }
        STR (HI(dx):2, buf2);   time_str := time_str + buf2;       { sec  }
        FOR i:=1 to 8 do begin
          IF (time_str[i] = ' ') then   time_str[i] := '0';
        end;
        Get_time := time_str;
     end;
  END;
```



```
{.pa}

function Get_Seconds : real;
  BEGIN
     WITH regs DO begin
        ah := $2C;
        MSDOS (regs);
        Get_Seconds := (((ch * 60.0) + cl)* 60.0) + dh + (dl / 100.0);
     end;
  END;
{.pa}

function Get_DATE : time_date_string;
  var  date_str  : time_date_string;
       buf4      : string[4];
       i         : byte;
  BEGIN
     WITH regs DO begin
        ax := $2A00;
        MSDOS (regs);
        STR (cx:4,      buf4);   date_str := buf4[3] + buf4[4] + '/'; { year }
        STR (HI(dx):2, buf4);    date_str := date_str + buf4 + '/';   { month}
        STR (LO(dx):2, buf4);    date_str := date_str + buf4;         { day  }
        FOR i:=1 to 8 do begin
          IF (date_str[i] = ' ') then   date_str[i] := '0';
        end;
        Get_Date := date_str;
     end;
  END;
{.pa}
```

```
{*}
{*source code copyright (c) 1986, TurboPower Software*}
{*}

{.E-}

FUNCTION BigTurboCardinal(i:Integer):Real;
 {-return a real in the range 0..65535}
VAR
 r:Real;
BEGIN
 r:=i;
 IF r<0 THEN r:=r+65536.0;
 BigTurboCardinal:=r;
END;{BigTurboCardinal}

FUNCTION ReturnFileInfo(ChnFileName:BigTurboString;
                        VAR F:FILE):Integer;
 {-error check and return file size and opened file var for the named file}
VAR
 size:Integer;

 FUNCTION ExistFile(fname:BigTurboString;VAR fullname:BigTurboString)
  :Boolean;
  {-search for the file, return true and a full pathname if found}
 CONST
  MaxPathEntries=10;
 TYPE
  PathArray=ARRAY[1..MaxPathEntries] OF BigTurboString;
 VAR
  F:FILE;
  slashpos:Byte;
  pathnum:Integer;
 CONST
  pathes:PathArray=
  ('','','','','','','','','','');
  bestpath:Integer=0;

  PROCEDURE GetPath;
   {-access the DOS environment and return a parsed array of path strings}
  VAR
   pathCnt,envSeg,envOfs:Integer;
   teststring:STRING[5];
   endOfEnv:Boolean;
   Ch:Char;
  BEGIN{GetPath}
   {get the address of the environment}
   envSeg:=MemW[CSeg:$2C];
   envOfs:=0;
   {scan the environment to find the PATH entry, if any}
   teststring[0]:=#5;
   REPEAT
    Move(Mem[envSeg:envOfs],teststring[1],5);
    endOfEnv:=(Copy(teststring,1,2)=#0#0);
    envOfs:=Succ(envOfs);
   UNTIL endOfEnv Oh (teststring='PATH=');
   IF teststring='PATH=' THEN BEGIN
    {parse the pathnames}
    envOfs:=envOfs+4;
    pathCnt:=1;
    REPEAT
     Ch:=Chr(Mem[envSeg:envOfs]);
     IF (Ch<>';') AND (Ch<>#0) THEN
      pathes[pathCnt]:=pathes[pathCnt]+Ch
     ELSE BEGIN
      {assure that all pathes end with \}
      IF pathes[pathCnt][Length(pathes[pathCnt])]<>'\' THEN
       pathes[pathCnt]:=pathes[pathCnt]+'\';
      pathCnt:=Succ(pathCnt);
```



```
     END;
     envOfs:=Succ(envOfs);
    UNTIL (Ch=#0) OR (pathCnt>MaxPathEntries);
   END;
  END;{GetPath}

 BEGIN{ExistFile}
  {see if the file exists as named}
  Assign(F,fname);
  {$I-}Reset(F){$I+};
  IF IOResult=0 THEN BEGIN
   fullname:=fname;
   ExistFile:=True;
   Close(F);
   Exit;
  END;

  {see if fname is already a pathname or includes an explicit drive}
  slashpos:=Pos('\',fname)+Pos(':',fname);
  IF slashpos<>0 THEN BEGIN
   {cannot prepend path names, we have failed}
   ExistFile:=False;
   Exit;
  END;

  {try the DOS environment PATH}
  IF pathes[1]='' THEN GetPath;

  IF bestpath<>0 THEN BEGIN
   {try the path that worked last time first}
   fullname:=pathes[bestpath]+fname;
   Assign(F,fullname);
   {$I-}Reset(F){$I+};
   IF IOResult=0 THEN BEGIN
    ExistFile:=True;
    Close(F);
    Exit;
   END;
  END;

  {try all the paths in order}
  pathnum:=1;
  WHILE (pathes[pathnum]<>'') AND (pathnum<=MaxPathEntries) DO BEGIN
   fullname:=pathes[pathnum]+fname;
   Assign(F,fullname);
   {$I-}Reset(F){$I+};
   IF IOResult=0 THEN BEGIN
    ExistFile:=True;
    Close(F);
    {store the path that worked for next time}
    bestpath:=pathnum;
    Exit;
   END;
   pathnum:=Succ(pathnum);
  END;

  {we didn't find file on path either}
  ExistFile:=False;
  fullname:=fname;

 END;{ExistFile}

BEGIN{ReturnFileInfo}
 {make sure it exists and open it}
 IF NOT(ExistFile(ChnFileName,ChnFileName)) THEN BEGIN
  WriteLn(Con);
  WriteLn(Con,'Module Object File ',ChnFileName,' not found....');
  Halt(1);
 END;
 Assign(F,ChnFileName);
```

```
 {note using a block size of 1}
 {$I-}Reset(F,1){$I+};

 {make sure there is something in it}
 size:=FileSize(F);
 IF size=0 THEN BEGIN
  WriteLn(Con);
  WriteLn(Con,'Module File ',ChnFileName,' is empty....');
  Halt(1);
 END;
 ReturnFileInfo:=size;
END;{ReturnFileInfo}

FUNCTION GetModuleRAM(size,ModNum:Integer):PointerType;
 {-error check and allocate memory for a module, returning a pointer}
VAR
 p:PointerType;
BEGIN
 {make sure there is space for it}
 IF (MaxAvail>=0) AND (size SHR 4>MaxAvail-32) THEN BEGIN
  WriteLn(Con);
  WriteLn(Con,'Module ',ModNum,' is too large to load on heap....');
  Halt(1);

 END;
 {allocate memory on heap}
 GetMem(p,size);
 GetModuleRAM:=p;
END;{getmoduleRAM}

PROCEDURE InitializeMainModule;
 {-set the pointers for the main module}
 {-should be called directly at the beginning of a chained main module}
CONST
 jmpset:JumpRecord=(offset:0;segment:0);
BEGIN
 ModuleIncomingPtr[0].offset:=Ofs(FarIncoming);
 ModuleSetupJumpPtr[0].offset:=Ofs(SetupJumpTable);
 {set up the local jump table}
 {use a far call to allow source code consistency in SetupJumpTable}
 jmpset:=ModuleSetupJumpPtr[0];
 INLINE(
  $2E/$FF/$1E/jmpset{CALL    FAR CS:jmpset}
 );
END;

{global variables private to this module}
CONST
 firstTime:Boolean=True;
 loadingOverlay:Boolean=False;
 GlobalOverlayNum:Integer=0;

 PROCEDURE InitializeModules;
  {-set up the main module and others the first time this is called}
 VAR
  m:Integer;

 BEGIN

  {check for valid call & set flags}
  IF firstTime THEN
   firstTime:=False
  ELSE
   Exit;

  {store the addresses of the main module}
  ModuleIncomingPtr[0].segment:=CSeg;
  ModuleSetupJumpPtr[0].segment:=CSeg;
  IF LoadDataIsEXEfile THEN Exit;
```



```
  InitializeMainModule;

  {initialize the local stack pointer}
  LocalStackPtr:=MaxStackSize;
  MinLocalStackPtr:=MaxStackSize;

  {initialize the pointers to all other modules}
  FOR m:=1 TO MaxNumModules DO
   WITH ModuleIncomingPtr[m] DO BEGIN
    segment:=0;
    offset:=0;
   END;
  FOR m:=0 TO MaxNumOverlays DO
   WITH OverlayArea[m] DO BEGIN
    size:=0;
    CurrentModule:=-1;
   END;
 END;{initializemodules}

 PROCEDURE LoadModule(ModNum:Integer;ChnFileName:BigTurboString);
  {-load the CHN File and set up the required addresses}
 CONST
  id1string:BigTurboString='FARCALLHANDLER FOLLOWS';
  id2string:BigTurboString='SETUPJUMPTABLE FOLLOWS';
  jmpset:JumpRecord=(ofs(set:0;segment:0);
 VAR
  F:FILE;{used to load the CHN file}
  i:Integer;
  FirstRunJump:Integer ABSOLUTE CSeg:$101;
  RuntimeSize:Integer;
  FileSize:Integer;
  moduleseg:Integer;
  ext:STRING[4];
  NumberToClear:Integer;

  FUNCTION SearchModule(idstring:BigTurboString;startofs:Integer):Integer;
   {-find the specified string in the current module image}
  VAR
   TestLen:Byte;
   teststring:^BigTurboString;
   TestChar:Char;
   firstord:Byte;
   i:Integer;
   foundit:Boolean;
  BEGIN
   i:=startofs;
   firstord:=Ord(idstring[0]);
   TestLen:=Succ(firstord);
   foundit:=False;
   REPEAT
    i:=Pred(i);
    IF Mem[moduleseg:i]=firstord THEN BEGIN
     teststring:=Ptr(moduleseg,i);
     {Move(Mem[moduleseg:i],teststring,TestLen);}
     foundit:=(teststring^ =idstring);
    END;
   UNTIL (i=0) OR foundit;
   IF i=0 THEN BEGIN
    WriteLn(Con);
    WriteLn(Con,'ID string ',idstring,' not found in Module ',ChnFileName);
    Halt(1);
   END;
   SearchModule:=i;
  END;{searchmodule}

 BEGIN{LoadModule}

  {assure the module number is legal}
  IF (ModNum>MaxNumModules) OR (ModNum<0) THEN BEGIN
   WriteLn(Con);
```

```
  WriteLn(Con,'Module number ',ModNum,' out of range');
  Halt(1);
 END;

 IF firstTime THEN
  InitializeModules;

 IF LoadDataIsEXEfile AND NOT(loadingOverlay) THEN BEGIN
  {we are running from a preinitialized EXE image}
  {get addresses back from EXE loader data}
  moduleseg:=LoadDataRelSeg[ModNum];
  ModuleSetupJumpPtr[ModNum].segment:=moduleseg;
  ModuleIncomingPtr[ModNum].segment:=moduleseg;
  ModuleSize[ModNum]:=LoadDataRelSize[ModNum];
  {reserve heap space that is already initialized}
  GetMem(ModulePtr[ModNum],ModuleSize[ModNum]);
  {copy the current PSP to the extra module}
  Move(Mem[CSeg:0],Mem[moduleseg:0],$100);
  Exit;
 END;

 {else we are loading extra modules at runtime}
 {first assure it is a .CHN file}
 ext:=Copy(ChnFileName,Length(ChnFileName)-3,4);
 FOR i:=1 TO 4 DO ext[i]:=UpCase(ext[i]);
 IF ext<>'.CHN' THEN BEGIN
  WriteLn(Con);
  WriteLn(Con,'Module File ',ChnFileName,' must be a .CHN file');
  Halt(1);
 END;

 {check for module # already loaded -- ASSUMES UNIQUE MODULE #s}
 WITH ModuleIncomingPtr[ModNum] DO
  IF (segment<>0) OR (offset<>0) THEN Exit;

 {find file and get the file size}
 FileSize:=ReturnFileInfo(ChnFileName,F);

 {store the file name}
 loaddatafilename[ModNum]:=ChnFileName;

 {get size of runtime library in bytes, plus its PSP}
 RuntimeSize:=FirstRunJump+$103;
 {store size of the module in RAM - $20 extra for paragraph alignment plus slop}
 ModuleSize[ModNum]:=RuntimeSize+FileSize+$20;

 IF loadingOverlay THEN WITH OverlayArea[GlobalOverlayNum] DO BEGIN

  {make sure the file fits in the overlay area}
  IF BigTurboCardinal(ModuleSize[ModNum])>BigTurboCardinal(size) THEN BEGIN
   WriteLn(Con);
   WriteLn(Con,'Overlay module ',ModNum,' larger than overlay area ',GlobalOverlayNum);
   Halt(1);
  END;

  {assign the pointer}
  ModulePtr[ModNum]:=pointer;
  moduleseg:=overlayseg;

  (* this step is no longer required
  {clear garbage from previous overlays}
  NumberToClear:=size-modulesize[modnum];
  IF NumberToClear<>0 THEN
  FillChar(Mem[moduleseg:(RuntimeSize+FileSize)],NumberToClear,0);
  *)

 END ELSE BEGIN

  {error check and allocate heap space}
  ModulePtr[ModNum]:=GetModuleRAM(ModuleSize[ModNum],ModNum);
```



```
  {add one to the pointer segment to guarantee starting on 0 offset boundary}
  moduleseg:=Succ(Seg(ModulePtr[ModNum]^));

  {copy runtime library to heap}
  Move(Mem[CSeg:0],Mem[moduleseg:0],RuntimeSize);

 END;

 {load .CHN code onto heap}
 BlockRead(F,Mem[moduleseg:RuntimeSize],FileSize);
 Close(F);

 {store the pointers}
 ModuleSetupJumpPtr[ModNum].segment:=moduleseg;
 ModuleIncomingPtr[ModNum].segment:=moduleseg;

 {search the far code for the idstrings identifying key offsets}
 {FarIncoming Procedure}
 i:=SearchModule(id1string,FileSize+RuntimeSize);
 ModuleIncomingPtr[ModNum].offset:=i-7;
 {SetupJumpTable Procedure}
 i:=SearchModule(id2string,i);
 ModuleSetupJumpPtr[ModNum].offset:=i-7;

 {set up the far jump table
 jmpset:=ModuleSetupJumpPtr
 INLINE(
  $2E/$FF/$1E/jmpset{CALL
  });

END;{LoadModule}

PROCEDURE UnloadModule(ModNum:Integer);
 {-release heap memory of module ModNum}
 {-if modnum is loaded as an overlay, just deassign it from the overlay area}
VAR
 n:Integer;
 isoverlay:boolean;
BEGIN
 isoverlay:=false;

 FOR n:=0 TO MaxNumOverlays DO
  WITH OverlayArea[n] DO
   IF ModNum=CurrentModule THEN BEGIN
    CurrentModule:=-1;
    isoverlay:=true;
   END;

 WITH ModuleIncomingPtr[ModNum] DO
  IF (segment<>0) OR (offset<>0) THEN BEGIN
   {nil the pointer so CloneCodeSegment knows what's up}
   segment:=0;
   offset:=0;
   {nil the file name}
   loaddatafilename[ModNum]:='';
   {free up the heap space}
   if not(isoverlay) then
    FreeMem(ModulePtr[ModNum],ModuleSize[ModNum]);
  END;
END;{UnloadModule}

PROCEDURE ReleaseOverlayArea(OverlayNum:Integer);
 {-dispose of the BIGTURBO overlay area}
BEGIN
 WITH OverlayArea[OverlayNum] DO
  IF size<>0 THEN BEGIN
   FreeMem(pointer,size);
   {nil the incoming pointer to the current module so CloneCodeSegment knows what's up}
   IF CurrentModule>0 THEN
```

```
    WITH ModuleIncomingPtr[CurrentModule] DO BEGIN
     segment:=0;
     offset:=0;
     {nil the file name}
     loaddatafilename[CurrentModule]:='';
    END;
   size:=0;
   CurrentModule:=-1;
  END;
END;{releaseOverlayArea}

PROCEDURE GetOverlayArea(OverlayNum:Integer;ChnFileName:BigTurboString);
 {-allocate a module area for overlaying modules to use}
 {-based on size of ChnFileName}
VAR
 F:FILE;
 FirstRunJump:Integer ABSOLUTE CSeg:$101;
 FileSize:Integer;
 RuntimeSize:Integer;
 NewSize:Integer;
BEGIN

 IF (OverlayNum>MaxNumOverlays) OR (OverlayNum<0) THEN BEGIN
  WriteLn('Overlay Area number ',OverlayNum,' out of range');
  Halt(1);
 END;

 IF firstTime THEN
  InitializeModules;

 {get size of runtime library in bytes, including PSP}
 RuntimeSize:=FirstRunJump+$103;

 {find file and get the file size}
 IF ChnFileName<>'' THEN BEGIN
  FileSize:=ReturnFileInfo(ChnFileName,F);
  Close(F);
 END ELSE
  {special case - allocate maximum area when null file is specified}
  FileSize:=$FFD0-RuntimeSize;

 WITH OverlayArea[OverlayNum] DO BEGIN
  NewSize:=RuntimeSize+FileSize+$20;
  IF BigTurboCardinal(size)>=BigTurboCardinal(NewSize) THEN
   Exit;
  IF size<>0 THEN
   ReleaseOverlayArea(OverlayNum);
  size:=NewSize;
  pointer:=GetModuleRAM(size,-OverlayNum);
  overlayseg:=Succ(Seg(pointer^));
  CurrentModule:=-1;
  {copy runtime library to heap}
  Move(Mem[CSeg:0],Mem[overlayseg:0],RuntimeSize);
 END;
END;{GetOverlayArea}

PROCEDURE LoadOverlay(OverlayNum,ModNum:Integer;ChnFileName:BigTurboString);
 {-put a BIGTURBO overlay into the predefined overlay area}
VAR
 n:Integer;
BEGIN

 {check for module # already loaded -- ASSUMES UNIQUE MODULE #s}
 WITH ModuleIncomingPtr[ModNum] DO
  IF (segment<>0) OR (offset<>0) THEN Exit;

 WITH OverlayArea[OverlayNum] DO BEGIN
  IF size=0 THEN
   GetOverlayArea(OverlayNum,ChnFileName);
  {nil the pointer of the current module so CloneCodeSegment knows what's up}
```



```
   IF CurrentModule>0 THEN
    WITH ModuleIncomingPtr[CurrentModule] DO BEGIN
     segment:=0;
     offset:=0;
    END;
   CurrentModule:=ModNum;
  END;
  loadingOverlay:=True;
  GlobalOverlayNum:=OverlayNum;
  LoadModule(ModNum,ChnFileName);
  loadingOverlay:=False;
END;{loadOverlay}

PROCEDURE BuildEXEifRequested;
 {-sets up LoadDataArea in code segment and clones the memory image to disk}
VAR
 n:Integer;
 F:FILE;
 namef:Text;
 bytes:Real;
 Sectors:Integer;
 MaxSeg,MaxSize:Integer;
 SectorsToWrite,SegNum,SecNum:Integer;
BEGIN
 IF NOT(LoadDataCloneMyself) THEN Exit;
 LoadDataMaxNumModules:=MaxNumModules;
 MaxSeg:=0;
 {find the highest module loaded on the heap}
 FOR n:=1 TO MaxNumModules DO
  WITH ModuleIncomingPtr[n] DO
   IF segment<>0 THEN BEGIN
    {segment offset relative to start of executable code (no PSP)}
    LoadDataRelSeg[n]:=segment-CSeg-$10;
    LoadDataRelSize[n]:=ModuleSize[n];
    IF BigTurboCardinal(LoadDataRelSeg[n])>BigTurboCardinal(MaxSeg) THEN BEGIN
     MaxSeg:=LoadDataRelSeg[n];
     MaxSize:=LoadDataRelSize[n];
    END;
   END;
 {assure proper initialization when EXE starts up}
 firstTime:=True;
 LoadDataIsEXEfile:=True;
 LoadDataCloneMyself:=False;
 {write the memory image}
 {calculate how many bytes to write}
 bytes:=16.0*BigTurboCardinal(MaxSeg)+BigTurboCardinal(MaxSize);
 Sectors:=1+Trunc(bytes/512.0);
 {write 'em}
 {$I-}
 Assign(F,'BUILDEXE.TMP');
 Rewrite(F,512);
 IF IOResult<>0 THEN BEGIN
  WriteLn('could not create EXE memory image file BUILDEXE.TMP');
  WriteLn('directory full or drive not ready');
  Halt(1);
 END;
 SegNum:=CSeg+$10;{don't write the PSP}
 SecNum:=0;
 WHILE SecNum<Sectors DO BEGIN
  SectorsToWrite:=8;
  IF SectorsToWrite>Sectors-SecNum THEN SectorsToWrite:=Sectors-SecNum;
  BlockWrite(F,Mem[SegNum:0],SectorsToWrite);
  IF IOResult<>0 THEN BEGIN
   WriteLn('could not write to BUILDEXE.TMP. Disk full or drive not ready.');
   Close(F);
   Erase(F);
   Halt(1);
  END;
  SecNum:=SecNum+SectorsToWrite;
  SegNum:=SegNum+$20*SectorsToWrite;{32 paragraphs per sector}
```

```
 END;
 Close(F);
 {write the object file names}
 Assign(namef,'BUILDEXE.NAM');
 Rewrite(namef);
 IF IOResult<>0 THEN BEGIN
  WriteLn('could not create object name file BUILDEXE.NAM');
  WriteLn('directory full or drive not ready');
  Halt(1);
 END;
 FOR m:=1 TO MaxNumModules DO BEGIN
  WriteLn(namef,loaddatafilename[m]);
  IF IOResult<>0 THEN BEGIN
   WriteLn('could not write to object name file BUILDEXE.NAM');
   Halt(1);
  END;
 END;
 Close(namef);
 {$I+}
 Halt(0);
END;{buildexeifrequested}

{.E+}
```



```
procedure RAM_Page( pageHandle : integer);          external 'ram-page.bin';

procedure Create_Page( var pageHandle : integer;
                           width,
                           height     : integer;
                       var result     : integer);   external Ram_Page[3];

procedure View_Page(     pageHandle: integer;
                     var x1,
                         y1,
                         x2,
                         y2 : integer);             external Ram_Page[6];

procedure Position_Page(     pageHandle: integer;
                         var xPos      : integer;
                         var yPos      : integer); external Ram_Page[9];

procedure Appear_View( pageHandle : integer);       external Ram_Page[12];

procedure Disappear_View( pageHandle : integer);    external Ram_Page[15];

procedure Frame_View( pageHandle  : integer;
                      frameStyle,
                      attribute   : byte);          external Ram_Page[18];

procedure Init_Ram_Page;                            external Ram_Page[21];

procedure Disable_View( pageHandle : integer);      external Ram_Page[24];

procedure Enable_View( pageHandle  : integer);      external Ram_Page[27];

procedure Copy_Window( pageHandle,
                       xPos,
                       yPos : integer);             external Ram_Page[30];
```

RWINDOW.INC


```
{=============================================================}
{ Written by   : Alex faui                                    }
{ first version: Sept 15, 86                                  }
{ Last Update  : 12:18:07  1/29/1987                          }
{=============================================================}

const   k_up    = 18432;  s_up    = 56;
        k_down  = 20480;  s_down  = 50;
        k_left  = 19200;  s_left  = 52;
        k_right = 19712;  s_right = 54;

        k_home  = 18176;
        k_end   = 20224;
        k_PgUp  = 18688;
        k_PgDn  = 20736;

        k_f1 = 15104;  k_f2 = 15360;  k_f3 = 15616;  k_f4 = 15872;
        k_f5 = 16128;  k_f6 = 16384;  k_f7 = 16640;  k_f8 = 16896;
        k_f9 = 17152;  k_f10= 17408;

type    RamPageRec = record
          name                      : string[20];
          handle,
          width, height,
          x1, y1, x2, y2,
          xView, yView, xOfs, yOfs,
          foreGround, backGround,
          xPos, yPos                : integer;
          frame_byte                : integer;
        end;
        Windows_Rec = Array[Windows] of RamPageRec;

var     Wresult                     : integer;
        W                           : Windows_Rec;
        ChgScr                      : RamPageRec;

        i, ii                       : integer;


var    W_recs : array[Windows] of Window_Limits_Rec absolute W;

{.pa}

procedure Init_Window_Limits;
  var setupFile : text;
      i         : byte;

  begin
    Assign (setupFile, 'cada-cta.def');
    {$i-} Reset (setupFile);  {$I+}
    if (IOresult = 0) then
      for i:=0 to num_of_windows-1 do begin
        with W[Windows(i)] do begin
          readln (setupFile, name);
          readln (setupFile, width, height);
          readln (setupFile, x1,    y1,    x2,   y2);
          readln (setupFile, xView, yView, xOfs, yOfs);
          readln (setupFile, foreground, background);
          readln (setupFile, xPos,  yPos);
          readln (setupFile, frame_byte);
        end;
      end
    else
      W_recs := Window_Limits;
    close (setupFile);
  end;

type  screen_type = record
                      ch    : char;
                      attr  : byte;
```



```
                    end;
var   screen      : array[1..25, 1..80] of screen_type absolute $b800:$0000;

procedure Appear_Window (NameW : RamPageRec);
var  len,i : integer;

begin
  with NameW do begin
    appear_view (handle);

    if (frame_byte = $ff) then begin
      len   := length(name);

      if ((Xview > 2) and (len > 0)) then begin
        if (len > Xview-2) then
          len := Xview - 2;
        for i:=1 to len do
          screen[Y1-1, X1+i].ch := name[i];
      end;
    end;
  end;
end;

{.pa}

procedure Init_Window (NameW      : Windows;
                       appear     : boolean;
                       frame      : boolean);
  begin
    with W[NameW] do begin
        Create_Page (handle, width, height, Wresult);
        if Wresult <> 0 then begin
            writeln ('Create_Page failed, result = ', Wresult);
            halt;
        end;

        xView := x2 - x1;     xOfs := width  - xView;
        yView := y2 - y1;     yOfs := height - yView;

        TextColor (foreGround);
        TextBackGround (backGround);

        ram_page (handle);
        clrscr;
        view_page  (handle, x1, y1, x2, y2);
        position_page (handle, xpos, ypos);
        if frame  then begin
            frame_byte  := $ff;
            frame_view (handle,  2,  $2C);  { $2C => FG:light red, BG:green }
        end
        else frame_byte := $00;
        if appear then
            Appear_Window (W[NameW]);
    end;
  end;


procedure Init_chg_scr;
  begin
    with ChgScr do begin
      width := 80;          height := 2;
      xView := 79;          xOfs := width  - xView;
      yView :=  2;          yOfs := height - yView;
      x1 :=  1;             x2 := x1 + xView;
      y1 := 24;             y2 := y1 + yView;
      xpos := 1;            ypos := 1;
      foreground := black;
      background := white;
      with ChgScr do begin
        Create_Page (handle, width, height, Wresult);
```

```
        if Wresult <> 0 then begin
            writeln ('Create_Page failed, result = ', Wresult);
            halt;
        end;
        TextColor (foreGround);
        TextBackGround (backGround);
        ram_page (handle);
        clrscr;
        view_page  (handle, x1, y1, x2, y2);
        position_page (handle, xpos, ypos);
      end;
    end;
  end;

{.pa}

function Key( var inputValue: integer;
              var funcKey : boolean ): boolean;
  var regpack:  record
                  ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
                end;
      regByte:  array[ 1..20] of byte absolute regpack;

  begin
    with regpack do begin
      ax := 6 shl 8;
      dx := 255;
      MsDos(regpack);
      if regByte[ 19] and $40 = 0 then begin
        key      := true;
        FuncKey  := false;
        inputValue := regByte[ 1];                { no function key }
        if inputValue = 0 then begin
          funckey := true;
          ax := 6 shl 8;
          dx := 255;
          MsDos(regpack);
          inputValue := regByte[ 1] shl 8;        { function key    }
        end;
      end
      else Key := false;
    end;
  end;


function InKey( var inputValue: integer ) : boolean;
  var funcKey : boolean;
  begin
    while (Not (Key(inputValue, funcKey) )) do;     { wait for keypressed }
    InKey := funcKey;
  end;

{.pa}

type  chg_opts = (All, Scroll, Size, Move, Change, Save, Recall, Delete);

procedure Change_screen (NameW : Windows; opt : chg_opts);
  var FuncKey, more_chg, quit_chg                   : boolean;
      keyValue                                      : integer;
  procedure Check_scroll_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : ypos := ypos - 1;
          k_down  : ypos := ypos + 1;
          k_left  : xpos := xpos - 1;
          k_right : xpos := xpos + 1;
          k_PgUp  : ypos := ypos - yView + 1;
          k_PgDn  : ypos := ypos + yView - 1;
          k_home  : begin
```



```
                      xpos := 1;  ypos := 1;
                    end;
          k_end   : begin
                      xpos := 1;  ypos := height;
                    end;
        end;
      end;
    end;

  procedure Check_enlarge_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : y1 := y1 - 1;
          k_down  : y2 := y2 + 1;
          k_left  : x1 := x1 - 1;
          k_right : x2 := x2 + 1;
        end;
      end;
    end;

  procedure Check_reduce_key;
    begin
      with W[NameW] do begin
        case keyValue of
          s_up    : y2 := y2 - 1;
          s_down  : y1 := y1 + 1;
          s_left  : x2 := x2 - 1;
          s_right : x1 := x1 + 1;
        end;
      end;
    end;

  procedure Check_move_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : begin  y1 := y1 - 1;  y2 := y2 - 1;  end;
          k_down  : begin  y1 := y1 + 1;  y2 := y2 + 1;  end;
          k_left  : begin  x1 := x1 - 1;  x2 := x2 - 1;  end;
          k_right : begin  x1 := x1 + 1;  x2 := x2 + 1;  end;
        end;
      end;
    end;

  procedure Check_Margin (Var x : integer; min, max : integer);
    begin
      if (x < min) then x := min;
      if (x > max) then x := max;
    end;
{.pa}

  procedure Save_Recall_Setup (opt : chg_opts);
    var setupFile  : text;
        inputOK    : boolean;
        kVal,i     : integer;
    begin
      if (opt = All) then begin
        write('F1-Save/ F2-Recall/ F3-Delete WINDOW SETUP FROM DISK');
        gotoXY (71,2);
        write  (' F10=EXIT');
        repeat
          FuncKey := InKey(keyValue);
        until (FuncKey and (keyValue >= k_F1) and (keyValue <= k_F10));
      end
      else begin
        case opt of
          Save   : keyValue := k_F1;
          Recall : keyValue := k_F2;
          Delete : keyValue := k_F3;
```

```
      end;
    end;
    gotoxy(1,2); delLine;
    Assign (setupFile, 'window.def');
    case keyValue of
      k_F1 : begin
               Rewrite (setupFile);
               for i:=0 to num_of_windows-1 do begin
                 with W[Windows[i]] do begin
                   writeln (setupFile, name);
                   writeln (setupFile, width:10, height:10);
                   writeln (setupFile, x1:10,    y1:10,    x2:10,   y2:10);
                   writeln (setupFile, xView:10, yView:10, xOfs:10, yOfs:10);
                   writeln (setupFile, foreground:10, background:10);
                   writeln (setupFile, xPos:10,  yPos:10);
                   writeln (setupFile, frame_byte:10);
                 end;
               end;
               Close   (setupFile);
               write   ('SETUP SAVED :');
             end;
      k_F2 : begin
               {$I-} Reset (setupFile); {$I+}
               inputOK := (IoResult = 0);
               if not inputOK then begin
                 gotoxy(1,2); delLine;
                 write ('Setup file not found');
               end
               else begin
                 write ('LOADING WINDOW SETUP');
                 for i:=0 to num_of_windows-1 do begin
                   with W[Windows[i]] do begin
                     readln (setupFile, name);
                     readln (setupFile, width, height);
                     readln (setupFile, x1,    y1,    x2,   y2);
                     readln (setupFile, xView, yView, xOfs, yOfs);
                     readln (setupFile, foreground, background);
                     readln (setupFile, xPos,  yPos);
                     readln (setupFile, frame_byte );
                     position_page (handle, xpos, ypos);
                     view_page     (handle, x1, y1, x2, y2);
                   end;
                 end;
                 Close  (setupFile);
                 Appear_Window (W[NameW]);
               end;
             end;
      k_F3 : begin
               write   ('Please press ''Y'' to confirm the deletion');
               If ( (not (Inkey (kVal))) and (kVal = integer('Y')) ) then begin
                 writeln;
                 write   ('Deleting ...');
                 close   (setupFile);
                 erase   (setupFile);
               end
               else begin
                 writeln;
                 write   ('Deleting Aborted');
               end;
             end;
    end;
  end;


procedure Change_Frame (background_color, frameStyle, attr : byte);
  begin
    with W[NameW] do begin
      TextBackGround (background_color);
      disable_view   (handle);
      disappear_view (handle);
```



```
      Frame_View     (handle, frameStyle, attr);
      enable_view    (handle);
      Appear_Window  (W[NameW]);
    end;
  end;
{.pa}

  var scroll_scr, size_scr,
      setup_scr,  move_scr, change_scr              : boolean;
      win_ofs                                       : integer;

  begin
    win_ofs := ord (NameW);
    Change_Frame   (red, 1, 0);

    repeat
      with W[NameW] do begin
        scroll_scr := false;  move_scr  := false;
        size_scr   := false;  change_scr:= false;  setup_scr := false;
        quit_chg   := false;
        Appear_Window  (ChgScr);
        ram_page       (ChgScr.handle);
        TextColor      (W[time_window].foreground);
        TextBackGround (W[time_window].background);
        gotoxy( 1,1);  write ('ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿',
                              '³                                                                ³');
        TextColor      (ChgScr.foreground);
        TextBackGround (ChgScr.background);
        gotoxy(20,1);  write (' Currently Editting : ',Name:18, ' ');
        if (opt = All) then begin
          gotoxy( 1,2);  write   (' F5 = Scroll,',
                                  ' F6 = Size,',
                                  ' F7 = Move,',
                                  ' F8 = Next Window,',
                                  ' F9 = Setup,       ');
          gotoXY (71,2);
          write (' F10=EXIT');
          repeat
            FuncKey := InKey(keyValue);
          until (FuncKey and (keyValue >= k_F1) and (keyValue <= k_F10));
        end
        else begin
          case opt of
            Scroll : keyValue := k_F5;
            Size   : keyValue := k_F6;
            Move   : keyValue := k_F7;
            Change : keyValue := k_F8;
            Save   : keyValue := k_F9;
            Recall : keyValue := k_F9;
            Delete : keyValue := k_F9;
          end;
        end;

        gotoxy(1,2); DelLine;
        case keyValue of
          k_F5 : begin
                   scroll_scr := true;
                   write('SCROLL SCREEN : use Arrow, Home, End, PgUp, PgDn keys to scroll page')
                 end;
          k_F6 : begin
                   size_scr:= true;
                   write('SIZING WINDOW: use Arrow-ENLARGE / Shift_Arrow-REDUCE');
                 end;
          k_F7 : begin
                   move_scr  := true;
                   write('MOVE SCREEN : use arrow keys to move window');
                 end;
          k_F8 : begin
                   Change_Frame (background, 2, $2C);
```

```
                change_scr := true;
                win_ofs := win_ofs + 1;
                if (win_ofs >= num_of_windows) then begin
                  win_ofs := 0;
                  NameW := Windows(0)
                end
                else
                  NameW := Windows(win_ofs);
                Change_Frame (red, 1, 0);
              end;
        k_F9 : begin
                setup_scr := true;
                Save_Recall_Setup (opt);
                View_page    (handle,x1,y1,x2,y2);
                Position_page(handle,xpos,ypos);
                Frame_view   (handle,  2,  $2C);  { $2C => FG:light red, BG:green }
                Appear_Window(W[NameW]);
              end;
        k_F10: begin
                quit_chg   := true;
                disappear_view (ChgScr.handle);
                Change_Frame   (background, 2, $2C);
              end;
      end;
      gotoXY (71,2);
      write  (' F10=EXIT');

      if (not (change_scr) and not (setup_scr)) then begin
        if not (quit_chg) then begin
          Appear_Window    (W[NameW]);
          more_chg := true;
        end;

        while ((not quit_chg) and more_chg) do begin
          FuncKey := InKey(keyValue);
          if (keyValue = k_F10) then
            more_chg := false
          else begin
            if      (scroll_scr) then check_scroll_key
            else if (size_scr)   then begin
                                        check_enlarge_key;
                                        check_reduce_key;
                                      end
            else if (move_scr)   then check_move_key;

            Check_Margin (x1,  1, 79);
            Check_Margin (x2, x1, 80);

            Check_Margin (y1,  1, 24);
            Check_Margin (y2, y1, 25);

            xView := x2 - x1;  xOfs := width  - xView;
            yView := y2 - y1;  yOfs := height - yView;

            Check_Margin (xpos, 1, xOfs);
            Check_Margin (ypos, 1, yOfs);

            position_page(handle, xpos, ypos);
            view_page    (handle, x1, y1, x2, y2);
          end;
        end;
      end;
    end;
  until (quit_chg or (opt <> All));
  Disappear_view (ChgScr.handle);
end;
```



```
{=============================================================}
{                                                             }
{ PC-TECH JOURNAL,       MARCH 1986,       PROGRAMMING PRACTICES  }
{                                                             }
{ TITLE: TAKING COMMAND IN TURBO PASCAL                PAGE 161  }
{                                                             }
{=============================================================}
{                                                             }
{           SUB-PROCESS AND MEMORY MANAGEMENT LIBRARY         }
{           -----------------------------------------         }
{                                                             }
{        REQUIRE TWO FILE TO COMPILE THIS MODULE. THEY ARE    }
{                  DOS4AH.COM, DOS4BH.COM                     }
{                                                             }
{           A DEMO PROGRAM IS AVAILABLE, WHICH IS NAMED       }
{                        DOSCOM.PAS                           }
{                                                             }
{=============================================================}
{ ----- not used -------
type r8086 = record
               ax, bx, cx, dx, bp, si, di, ds, es, flags: integer;
             end;
var  regs  : r8086;
  ---------------------- }

type asciiz = string[ 65 ];

{ Current Drive                                               }
{-------------------------------------------------------------}
function    Current_Drive: integer;
  begin
    regs.ax := $1900;
    msdos(regs);
    Current_Drive := lo(regs.ax);
  end;


{ Current Directory                                           }
{-------------------------------------------------------------}
var d_str : asciiz;

function    Current_Dir (drive : integer;
                         Var Dir_str : asciiz): integer;
  var i     : byte ABSOLUTE dir_str;

  begin
    regs.ax := $4700;
    regs.ds := Seg(Dir_str);
    regs.si := Ofs(Dir_str[1]);
    regs.dx := drive;
    msdos(regs);
    if (regs.flags and 1) <> 0 then
        Current_Dir := lo(regs.ax)
    else begin
        { determine & set LENGTH of DIR_STR }
        i := 1;
        while ( (dir_str[i] <> chr(0)) and (i < SizeOf(dir_str)) )
            i := i + 1;
        i := i - 1;

        Current_Dir := 0;
    end;
  end;


{ Prompt with Current Drive & Dir                             }
{-------------------------------------------------------------}
function    dir_prompt (Var Prompt_str : asciiz): integer;
  begin
    dir_prompt := current_dir (0, prompt_str);
```

```
    prompt_str := chr(65 + current_drive) + ':\' + prompt_str;
  end;


{ Maximum MEMORY available                                              }
{----------------------------------------------------------------------}

function    Memory_Avail: integer;
    begin
      regs.es := Cseg;
      regs.ax := $4a00;
      regs.bx := $ffff;

      msdos(regs);
      Memory_Avail := regs.bx Div 64;
    end;


{ REDUCE MEMORY ALLOCATION                                              }
{----------------------------------------------------------------------}

function    dos4AH(pp_to_release: integer): integer;
    external 'spawn4AH.bin';


{ EXECUTE A SUB PROCESS                                                 }
{----------------------------------------------------------------------}
function    dos4BH(var program_name, parameter_string): integer;
    external 'spawn4BH.bin';


{ ALLOCATE A NEW MEMORY BLOCK                                           }
{----------------------------------------------------------------------}

function    dos48H(pp_needed: integer; var block_segment: integer): integer;

  begin
    regs.bx := pp_needed;                   { # of paragraphs req.    }
    regs.ax := $48 shl 8;                   { Function call 48H.      }
    msdos(regs);                            { Call DOS                }
    if (regs.flags and 1) <> 0 then begin{ Is carry flag set          }
        block_segment := regs.bx;           { Yes, return avail. pps  }
        dos48H        := lo(regs.ax);       { and error number        }
    end
    else begin
        block_segment := regs.ax;           { No, return seg. address}
        dos48H        := 0;                 { and error code 0.       }
        end;
    end;


{ RELEASE A MEMORY BLOCK                                                }
{----------------------------------------------------------------------}

function    dos49H(block_segment: integer): integer;

  begin
    regs.es := block_segment;       { segment address to release. }
    regs.ax := $49 shl 8;               { Func. call 49H.         }
    msdos(regs);                        { Call DOS.               }
    if (regs.flags and 1) <> 0 then     { Is carry flag set       }
        dos49H := lo(regs.ax)           { Yes, ret. error no.     }
    else
        dos49H := 0;                    { No, ret. error code 0   }
  end;


{ OBTAIN A PROCESS'S EXIT CODE                                          }
{----------------------------------------------------------------------}
```



```
function    dos4DH: integer;

  begin
    regs.ax := $4d shl 8;               { Function call 4dH       }
    msdos(regs);                        { Call DOS                }
    dos4DH  := lo(regs.ax);             { Return exit code        }
  end;


{ GET COMMAND PROCESSOR NAME                                            }
{----------------------------------------------------------------------}

function    get_comspec(var comspec: asciiz): boolean;

type
    dos_env_string =^dos_env_type;
    dos_env_type   =array [1..254] of byte;

var
    dos_env:    dos_env_string;
    dos_envs:   string [255];
    idx:        integer;

  begin
    get_comspec   := false;
    dos_env       := ptr(memw[cseg:$2c], $0);       { Get 254 bytes of the  }
    move(dos_env^, dos_envs[1], 254);               { DOS environ. string   }
    dos_envs[255] := #0;
    dos_envs[0]   := #255;
    idx           := pos('COMSPEC=', dos_envs);     { Find COMSPEC= portion }
    if idx= 0 then begin
        writeln('*** "COMSPEC=d:[path]filename" not in DOS.');
        get_comspec := true;
        exit;
    end
    else begin
        delete(dos_envs, 1, idx+7);                 { Isolate ASCIIZ string }
        idx      := pos(#0, dos_envs);              { drive[path]filename   }
        dos_envs := copy(dos_envs, 1, idx);         { of com. processor     }
        while dos_envs[1]=' ' do
            delete(dos_envs, 1, 1);
        comspec  := dos_envs;                       { Return FALSE          }
    end;
  end;


{ HANDLE A DOS ERROR CONDITION                                          }
{----------------------------------------------------------------------}

function    dos_error_check(error_code: integer): boolean;

    type
        error_table_type= array[1..18] of string[41];

    const
        error_table: error_table_type=    { RANGE: 1 TO 18 DEC.  }
            ('Invalid function number',
             'File not found',
             'Path not found',
             'Too many open files (no handles left)',
             'Access denied',
             'Invalid file handle',
             'Memory control blocks destroyed',
             'Insufficient memory',
             'Invalid memory block address',
             'Invalid environment',
             'Invalid format',
             'Invalid access code',
             'Invalid data',
             'UNRECOGNIZED ERROR',
```

```
          'Invalid drive was specified',
          'Attempted to remove the current directory',
          'Not same device',
          'No more files');

begin
  dos_error_check := true;
  if error_code= 0 then
     dos_error_check := false
  else
     writeln('*** DOS error ', error_code, ': ', error_table[error_code]);
end;
```



```
type  st_type = string[255];

function B_str (B_num : byte; len : byte): st_type;
var   st      : st_type;
begin
  str (B_num : len, st);  B_str := st;
end;


function I_str (I_num : integer; len : byte): st_type;
var   st      : st_type;
begin
  str (I_num : len, st);  I_str := st;
end;


function R_str (R_num : real; mantissa, decimal : byte): st_type;
var   st      : st_type;
begin
  str (R_num : mantissa : decimal, st);  R_str := st;
end;

{.pa}

function S_str (S_st        : st_type;          { ASIII string  }
                L_or_C_or_R : char;             { Left, Center or Right justify }
                len         : byte;             { length of output string }
                fill_ch     : char):    st_type; { fill charactor }
var i,       s_len,
    b_start, b_end,
    s_start, s_end   : byte;
    start            : integer;
    st               : st_type;
begin
  s_len := length (S_st);
  start := len - s_len;
  if ( (start < 0) or (s_len >= 80) or (len >= 80) ) then begin
    st[0] := chr(len);
    for i:=1 to len do
      st[i] := Fill_Ch;
  end
  else begin
    if (upcase(L_or_C_or_R) = 'L') then begin
      s_start := 1;                 s_end := s_len;
      b_start := s_end + 1;         b_end := len;
    end
    else if (upcase(L_or_C_or_R) = 'C') then begin
      b_start := 1;                 b_end := len;
      s_start := 1 + start div 2;   s_end := s_start + s_len - 1;
    end
    else begin
      b_start := 1;                 b_end := start;
      s_start := b_end + 1;         s_end := len;;
    end;

    st[0] := chr(len);              { set the length of the output string }
    for i:=b_start to b_end do      { file the non-used spaces with fill char }
      st[i] := Fill_Ch;
    for i:= s_start to s_end do     { copy the source string to the output str}
      st[i] := S_st[i - s_start + 1];
  end;

  S_str := st;
end;

{.pa}

Var DispV_BG,               { original  background color }
    DispV_FG,               {           foreground color }
    DispV_BG1,              { new msg   background color }
```

```
    DispV_FG1,              {           foreground color }
    DispV_BG2,              { new value background color }
    DispV_FG2 : integer;    {           foreground color }

procedure DispValue (Msg : st80; Vtype : char; Var Value; len, dec: integer);
var Bvalue : byte    absolute Value;
    Ivalue : integer absolute Value;
    Rvalue : real    absolute Value;
    Str    : st80    absolute Value;
begin
  write(' ');
  TextColor     (DispV_FG1);
  TextBackGround(DispV_BG1);
  write(Msg);
  TextColor     (DispV_FG2);
  TextBackGround(DispV_BG2);
  case Vtype of
    'B' : write(Bvalue:len);
    'I' : write(Ivalue:len);
    'R' : write(Rvalue:len:dec);
    'S' : write(Str   :len);
  end;
  TextColor     (DispV_FG);
  TextBackGround(DispV_BG);
end;
```



```
{ .cc$027&1C0$027(0U$027(s$p16.66h7v0s0b8T$027&18D$027&176F)
{.he********************************** BIGTURBO.VAR page # ********************************
******}
{.fo********** Copyright (c) 1986, TurboPower Software. All Rights Reserved **********}
{.pl76}
{.po10}
{.pw120}
{.in+}
{*}
{*}
{*source code copyright (c) 1985, by TurboPower Software*}
{*}
{*}

CONST
 bigturCopyright:STRING[79]='BIGTURBO - Large Code Model. Copyright (c) 1985 by TurboPower Softw
are.';
 bigturVersion:STRING[79]='All Rights Reserved. Version 1.06D';
 {Turbo data area in code segment of each module}
 TurboRunDataStart=$103;
 TurboRunDataLength=$100;
 {overlay areas - change number for fewer or more module overlays}
 MaxNumOverlays=2;

TYPE
 {string used for pathname to far code files}
 BigTurboString=STRING[64];

 {used to hold pointers for long jumps}
 JumpRecord=
 RECORD
  offset,
  segment:Integer;
 END;

 {holds info describing module overlay areas}
 PointerType="Integer;
 OverlayAreaRecord=
 RECORD
  pointer:PointerType;
  size,
  overlayseg,
  currentmodule:Integer;
 END;

VAR
 {holds name of each object module - used by BUILDEXE and TMERGE}
 LoadDataFileName:array[1..maxnummodules] of bigturboat::ing;

 {holds size allocated for each extra code segment}
 ModuleSize:ARRAY[0..MaxNumModules] OF Integer;

 {holds pointers to far code storage}
 ModulePtr:ARRAY[0..MaxNumModules] OF PointerType;

 {holds pointers to incoming call handlers}
 ModuleIncomingPtr:ARRAY[0..MaxNumModules] OF JumpRecord;

 {holds pointers to SetupJumpTable routines in far modules}
 ModuleSetupJumpPtr:ARRAY[0..MaxNumModules] OF JumpRecord;

 {holds descriptive information regarding reserved overlay areas}
 OverlayArea:ARRAY[0..MaxNumOverlays] OF OverlayAreaRecord;

 {an isolated stack for far calls}
 LocalStack:ARRAY[0..MaxStackSize] OF Byte;
 {stack pointer}
 LocalStackPtr:Integer;
 {keeps track of minimum value of stack pointer}
 MinLocalStackPtr:Integer;
```

```
{another stack holding debug information}
DebugStack:ARRAY[0..MaxStackSize] OF Byte;
{temporary variables for debug error handling}
tError,tSegnum,tProcnum:Integer;

{$-}

PROCEDURE CloneCodeSegment(startadd,bytes:Integer);
 {-copy a section of the current code segment to all other modules}
VAR
 m:Integer;
BEGIN
 FOR m:=0 TO MaxNumModules DO
  WITH ModuleIncomingPtr[m] DO
   IF NOT((segment=0) AND (offset=0)) THEN
    IF segment<>CSeg THEN
     Move(Mem[CSeg:startadd],Mem[segment:startadd],bytes);
END;                            {clonecodesegment}

FUNCTION currentmodule:Integer;
 {-return the currently active module}
VAR
 tmodule:Integer;
BEGIN
 IF LocalStackPtr<MaxStackSize THEN BEGIN
  Move(DebugStack[LocalStackPtr],tmodule,2);
  currentmodule:=tmodule;
 END ELSE
  currentmodule:=0;
END;                            {currentmodule}

PROCEDURE DumpFarCallStack;
 {-write a trace of the active intermodule calls}
VAR
 sp,tmodule,tproc,retseg,retofs:Integer;
TYPE
 HexString = string[4];

 FUNCTION Hex(i : Integer) : HexString;
   {-return hex representation of integer}
 CONST
   hc : ARRAY[0..15] OF Char = '0123456789ABCDEF';
 VAR
   l, h : Byte;
 BEGIN
   l := Lo(i); h := Hi(i);
   Hex := hc[h SHR 4]+hc[h AND $F]+hc[l SHR 4]+hc[l AND $F];
 END;                           {hex}

BEGIN
 sp:=LocalStackPtr;
 WriteLn;
 WHILE sp<MaxStackSize DO BEGIN
  Move(DebugStack[sp],tmodule,2);
  Move(DebugStack[sp+2],tproc,2);
  Move(LocalStack[sp],retseg,2);
  Move(LocalStack[sp+2],retofs,2);
  WriteLn('module: ',tmodule:2,' procedure: ',tproc:2,
          ' called from ',hex(retseg),':',hex(retofs));
  sp:=sp+6;
 END;
 WriteLn('module:  0');
END;                            {dumpfarcallstack}

PROCEDURE FarOutGoing(segnum,procnum:Integer);
 {-transfer control to the far segment - with error checking}
BEGIN
 INLINE(
```



```
 {get the Turbo parameters}
 $8B/$46/$04/                {MOV    AX,[BP+04] - store procnum in ax}
 $8B/$76/$06/                {MOV    SI,[BP+06] - get far segment number}

 {store the parameters to temporary global variables in case of error}
 $A3/tProcnum/               {MOV    tprocnum,AX}
 $89/$36/tSegnum/            {MOV    tsegnum,SI}

 {remove all but the far call parameters from the Turbo stack}
 $8B/$EC/$00/                {MOV    BP,[BP+00]}
 $8B/$E5/                    {MOV    SP,BP}

 {get the local stack pointer and assure room for this call}
 $8B/$3E/LocalStackPtr/      {MOV    DI,localstackptr}
 $81/$FF/$06/$00/            {CMP    DI,0006}
 $7F/$08/                    {JG     OK1:}

 {not enough stack space - store error code and jump to error hander}
 $C7/$06/tError/$01/$00/     {MOV    terror,0001}
 $EB/$5D/                    {JMP    err}

 {OK: - store what we must on local stack}
 $BB/LocalStack/             {MOV    BX,offset(localstack)}
 $81/$EF/$02/$00/            {SUB    DI,0002}
 $8F/$01/                    {POP    [BX+DI] - bp of calling proc}
 $81/$EF/$02/$00/            {SUB    DI,0002}
 $8F/$01/                    {POP    [BX+DI] - return offset}
 $81/$EF/$02/$00/            {SUB    DI,0002}
 $8C/$C9/                    {MOV    CX,CS}
 $89/$09/                    {MOV    [BX+DI],CX - return segment}
 $89/$3E/LocalStackPtr/      {MOV    localstackptr,DI -
                             store local stack ptr again}

 {store minimum value of local stack pointer}
 $3B/$3E/MinLocalStackPtr/   {CMP DI,minlocalstackptr}
 $7D/$04/                    {JGE   OK2}
 $89/$3E/MinLocalStackPtr/   {MOV minlocalstackptr,DI}

 {OK2:}
 {store the target module (segment) & procedure numbers on the debug stack}
 $BB/DebugStack/             {MOV    BX,offset(debugstack)}
 $89/$31/                    {MOV    [BX+DI],SI}
 $8B/$CE/tprocnum/           {MOV    CX,tprocnum}
 $89/$49/$02/                {MOV    2[BX+DI],CX}

 $BB/ModuleIncomingPtr/      {MOV    BX,offset(moduleincomingptr)}
 $D1/$E6/                    {SHL    SI,1}
 $D1/$E6/                    {SHL    SI,1 - convert module number to offset in pointer array}

 {check to see if pointer is initialized}
 $C7/$06/tError/$02/$00/     {MOV    terror,0002}
 $81/$38/$00/$00/            {CMP    WORD PTR [BX+SI],0000}
 $74/$19/                    {JZ     err}

 {check to see if the pointer points to a reasonable place}
 $C4/$38/                    {LES    DI,[BX+SI]}
 $C7/$06/tError/$03/$00/     {MOV    terror,0003}
 $26/$80/$7D/$01/$8B/        {CMP    ES:[DI+1],8B}

(*don't check the first byte as that may be overwritten by a breakpoint
 $26/$81/$3D/$55/$8B/        {CMP    ES:[DI],8B55}
 $90/$90/
*)

 $75/$0A/                    {JNZ    err}
 $26/$81/$7D/$02/$EC/$55/    {CMP    ES:[DI+02],55EC}
 $75/$02/                    {JNZ    err}

 $FF/$28                     {JMP    FAR [BX+SI] - go far}
 );
```

```
  {errors come here}
  CASE tError OF
   1:WriteLn(Con,'Out of BIGTURBO stack space or no modules loaded.');
   2:WriteLn(Con,'Attempting to call routine in uninitialized module.');
   3:WriteLn(Con,'Invalid FarIncoming pointer or corrupt module code.');
  ELSE
   WriteLn(Con,'Program error - code ',tError,' in FarOutgoing.');
  END;
  if terror<>1 then
    localstackptr:=localstackptr+6;
  WriteLn(Con,'Error while calling module: ',tSegnum,', procedure: ',tProcnum,
                ' from module ',CurrentModule);
  dumpfarcallstack;
  Halt(2);

END;                            {faroutgoing}

{.E+}
```



```
{.N cada_cv1}
{$V-}
{$K-}
{$R-}
Program CTA;
 { ------------------------------------------------------------------------}
 { Written by  : Alex Tsui                                                 }
 { Date        : Nov. 1, 1985                                              }
 { Last Update : 10/05/88                                                  }
 { ------------------------------------------------------------------------}
 { Title       : Interactive Controlling System for the DISA CTA/TRAVER    }
 { ------------------------------------------------------------------------}

 { ------------------------------------------------------------------------}
 { Procedure Debugging Aid                                                 }
 { ------------------------------------------------------------------------}
 {$I   ER-STACK.inc }

 { ------------------------------------------------------------------------}
 { DOS subprocess / LIBRARY call var. declaration                          }
 { ------------------------------------------------------------------------}
 type  regpack  = record case integer of
                    1: (ax,bx,cx,dx,bp,si,di,ds,es,flags: integer);
                    2: (al,ah,bl,bh,cl,ch,dl,dh         : byte);
                  end;
 var   regs     : regpack;
 var   bell     : char;

 {.pa}

 { ------------------------------------------------------------------------}
 { Window declaration                                                      }
 { ------------------------------------------------------------------------}
 const num_of_windows =7;
 type  Windows = (User, Input, Device, Sam, Help, Time_window, dummy);
 type  st20            = string[20];
       Window_Limits_Rec = record
                             N : st20;
                             A : Array[0..15] of Integer;
                           end;

 const  Window_Limits     : Array[Windows] of Window_Limits_Rec
          { Window Name }              {ble,  wi,he,   x1,y1,  x2,y2, xv,yv,xo,yo,   fG, bG,   x
pos,ypos, frame_byte}
        =((N: 'User Window ';         A: (0,  79,25,   2, 11,  38,23, 0, 0, 0, 0, black, green,
  1, 17,  255)),
          (N: 'Input Window ';        A: (0,  79,25,   41,11,  79,23, 0, 0, 0, 0, black, cyan,
  1, 21,  255)),
          (N: 'Device Window ';       A: (0,  79,25,   2, 3,   79,8,  0, 0, 0, 0, white, blue,
  1, 1 ,  255)),
          (N: 'SAE-SAM Window ';      A: (0,  15,15,   70, 3,  79,13, 0, 0, 0, 0, black, blue,
  1, 1 ,  255)),
          (N: 'Help Window ';         A: (0,  79,120,  2, 11,  79,23, 0, 0, 0, 0, white, black,
  1, 1 ,  255)),
          (N: 'Time & Date Window ';  A: (0,  80,1,    1, 1,   80,1,  0, 0, 0, 0, white, red,
  1, 1 ,  0)),
          (N: 'Dummy Window';         A: (0,   1,1,    1, 1,    1,1,  0, 0, 0, 0, white, red,
  1, 1 ,  0)));

 {.pa}

 {=========================================================================}
 { Command Interpreter Program                                             }
 {=========================================================================}
 { - - GLOBAL VARIABLES - - - - - - - - - - - - - - - - - - - - - - - - - - }

 CONST max_no_cmd = 25;
       ValidCmd_s : array[0..max_no_cmd] of string[3]
                  =('CRM', 'CME', 'CON',
                    'SRM', 'SME', 'SCO', 'SAM',
```

```
                    'TRA', 'PRO', 'LOA', 'ALL',
                    'DEB', 'DIS', 'DOS', 'DO:', 'ENA', 'END', 'EXE',
                  ' 'EXI', 'FIL', 'HEL', 'REA', 'QUI', 'WAI', 'WIN', 'BEL');

TYPE  ValidCmd_t =(c_rms, c_mean, c_cond,
                    s_rms, s_mean, s_corr, s_san,
                    c_trav,  c_probe,   c_lda, c_all,
                    c_debug, c_disable, c_dos, c_do,  c_enable,c_end, c_execute,
                    c_exit,  c_file,    c_help,c_read,c_quit,  c_wait,c_window, c_bell);

      st3       = string[3];
      st2       = string[2];
      st80      = string[80];

Type   dev_cpt      = (read_d, set_d, clear_d, init_d);
Const  dev_cpt_char : array[0..3] of char
                    = ('R', 'S', 'C', 'I');

var enable : record
               cta_rms  : boolean;
               cta_mean : boolean;
               cond     : boolean;
               sae_rms  : boolean;
               sae_mean : boolean;
               sae_san  : boolean;
               sae_corr : boolean;
               trav     : boolean;
               probe    : boolean;
             end;

Var   _help_loaded    : boolean;
      cmd_file        : array [0..10] of text;
      cmd_file_exist  : array [0..10] of boolean;
      cmd_num         : integer;

      DOS_mem         : real;

{.pa}

{ ------------------------------------------------------------------------}
{ System Utilities Routine. Required above tyte declaratics               }
{ ------------------------------------------------------------------------}
procedure printf (message : st128);
  begin
    write (message);
  end;

procedure printfln (message : st128);
  begin
    writeln (message);
  end;

{$I   ram-page.inc}           { Include:  Ram Page Window package   }
{$I   RMindow.inc }           { Include:  Window routine developed  }
                              {           from RamPage              }
{$I   spawn.inc   }
{$I   cli-dos.drv }           { Include:  routine to create subprocess }

{$I   Library.inc }
{$I   STRING.inc  }

{.pa}

{ ------------------------------------------------------------------------}
{ Window routines                                                         }
{ ------------------------------------------------------------------------}
VAR time,     date,
    old_time, old_date : time_date_string;
    td_ctr             : integer;
```



```
procedure update_Time_Date (immedate : boolean);
  var old_Row,  old_Col  : byte;
      mem : real;

  BEGIN
    time := Get_time;
    if (immedate or (time <> old_time)) then begin
      Appear_View (W[DUMMY].handle);
      old_time := time;
      date     := Get_date;
      mem := MemAvail;
      if (mem < 0) then mem := mem + 65536.0;
      mem := mem * 16.0 / 1024.0;
      with W[TIME_WINDOW] do begin
        Ram_page        (handle);
        Disable_view    (handle);
        TextColor       (foreground);
        TextBackGround  (background);
        GotoXY(3, 1);    printf ('Date : ' + Date);
        GotoXY(22,1);    printf ('Dos Avail: ' + R_str(Dos_mem,5,0) + 'K');
        GotoXY(42,1);    printf ('CTA Avail: ' + R_str(mem    ,5,0) + 'K');
        GotoXY(64,1);    printf ('Time : ' + Time);
        Enable_view     (handle);
        Appear_View     (W[TIME_WINDOW].handle);
        td_ctr := 0;
      end;
    end;
    td_ctr := td_ctr + 1;
  END;

Procedure Open_device_window;
  BEGIN
    Appear_View (W[DUMMY].handle);
    with W[DEVICE] do begin
      Ram_page       (handle);
      TextColor      (foreground);
      TextBackGround (backGround);
    end;
  END;

Procedure Open_SAM_window;
  BEGIN
    Appear_View (W[DUMMY].handle);
    with W[SAM] do begin
      Ram_page       (handle);
      TextColor      (foreground);
      TextBackGround (backGround);
    end;
  END;

Procedure Open_user_window;
  BEGIN
    Appear_View (W[DUMMY].handle);
    with W[USER] do begin
      Ram_page       (handle);
      Appear_Window  (W[USER]);
      TextColor      (foreground);
      TextBackGround (backGround);
    end;
  END;

Procedure Open_cmd_window;
  BEGIN
    Appear_View (W[DUMMY].handle);
    with W[INPUT] do beg.n
      Ram_Page       (handle);
      Appear_Window  (W[INPUT]);
      TextColor      (black);
      TextBackGround (backGround);
    end;
```

```
   END;

Procedure Open_Windows;
  BEGIN
     Window (1,1, 80,25);
     TextColor (white);
     clrscr;
     Init_ram_page;
     Init_chg_scr;
     Init_Window_Limits;

     Init_Window (time_window,    false,  false);
     Init_Window (device,         true,   true);
     Init_Window (user,           true,   true);     GotoXY (1,25);
     Init_Window (input,          true,   true);     GotoXY (1,25);
     Init_Window (Sam,            true,   true);
     Init_Window (help,           false,  true);
     Init_Window (Dummy,          false,  false);

     Update_Time_Date (true);
     Open_user_window;
  END;

Procedure Display_All_Windows;
  begin
    Appear_Window (W[TIME_WINDOW]);
    Appear_Window (W[DEVICE]);
    Appear_Window (W[USER]);
    Appear_Window (W[INPUT]);
    Appear_Window (W[SAM]);
  end;

{.pa}

{ ------------------------------------------------------------------------------}
{ Command Line Interpreter for the above drivers                                 }
{ ------------------------------------------------------------------------------}

Procedure Exit_to_Cli;   Forward;

Procedure AbortCLI (buf : st80);
  var  temp_ptr : Msg_ptr;
  BEGIN
     Push_msg (' ** ' + buf);
     Open_User_Window;

     printfln ('');
     printfln ('## possible cause of error ##');
     temp_ptr := first_msg;
     While (temp_ptr <> nil) do begin
       printfln (temp_ptr^.msg);
       temp_ptr := temp_ptr^.next;
     end;

     printfln ('Type HELP for more information' + Bell);
     Exit_to_Cli;
     Open_cmd_window;
     Pop_Msg;
  END; { AbortCLI }

Procedure DisplayError (err_code : byte);
  BEGIN
     CASE err_code OF
        1 : AbortCLI ('invalid data specified');
        2 : AbortCLI ('no data specified');
        3 : AbortCLI ('invalid sub_command');
        4 : begin
              printfln ('Invalid command' + Bell);
              printfln ('Type HELP for more info.');
              Exit_to_Cli;
```



```
            end;
     end
  END; { DisplayError }

{.pa}

Procedure UpdateWindow;   Forward;

{ ------------------------------------------------------------------------------}
{ National Instruments IEEE-488 interface                                        }
{ ------------------------------------------------------------------------------}
{.M cada_cvl }

{$I   dd-ni488.drv }                  { include : declactions for GPIB-PCIIA}
{$I   dd-gpib.drv  }


{ ------------------------------------------------------------------------------}
{ GPIB communication drivers                                                     }
{ ------------------------------------------------------------------------------}

{$I dd-cta.drv   }
{$I dd-sae.drv   }
{$I dd-sam.drv   }
{$I dd-trav.drv  }
{$I dd-probe.drv }

{$I cli-cta.drv  }
{$I cli-sae.drv  }
{$I cli-sam.drv  }
{$I cli-tra.drv  }
{$I cli-probe.drv}

{.pa}

{.M MainModule}

{ ------------------------------------------------------------------------------}
{ Window Display routine the above drivers                                       }
{ ------------------------------------------------------------------------------}
Procedure UpdateWindow;
  Const Bk    : st80 = '';

  VAR   i     : byte;
        buf,
        on_off: string[10];
        lines : byte;
  BEGIN
    Open_device_window;
    Disable_view (W[DEVICE].handle);
    ClrScr;

    lines := 0;
    DispV_BG  := W[DEVICE].background;
    DispV_BG1 := DispV_BG ; { Cyan; }
    DispV_BG2 := DispV_BG ; { Cyan; }

    DispV_FG  := W[DEVICE].foreground;
    DispV_FG1 := Yellow   ; { White; }
    DispV_FG2 := White    ;

    if (enable.Cta_rms) then with _Crms do begin
       lines := lines + 1;
       if OL then buf := 'overflow'
       else       str (data:9:5, buf);

       DispValue ('CTA-RMS ['+B_str(dev,2)+']('+B_str(cha,3)+')','S',Bk,0,0);
       DispValue ('  data=','S', buf, 9, 0);
       DispValue ('   tc=','R', tc,  8, 1);
       printfln ('');
```

```
    end;

    if (enable.Cta_mean) then with _Cmean do begin
       lines :  lines + 1;
       if OL then buf := 'overflow'
       else       str (data:9:5, buf);

       if filter then  on_off := 'ON'
       else            on_off := 'OFF';

       DispValue ('CTA-MEAN['+B_str(dev,2)+']['+B_str(cha,3)+']','S',Bk,0,0);
       DispValue ('  data=',  'S', buf,     9, 0);
       DispValue ('   tc=',   'R', tc,      8, 1);
       DispValue ('filt.=',   'S', on_off, 8, 0);
       printfln ('');
    end;

    if (enable.Cond) then begin
       FOR i := 0 to 2 DO with _Cond.cha[i] do begin
          if (disp) then begin
             lines := lines + 1;
             DispValue ('CTA-COND['+B_str(_Cond.dev,2)+']['+B_str(i,3)+']','S',Bk,0,0);
             DispValue ('  gain=', 'R', gain,9,1);
             DispValue ('   LP=', 'R', lp,8,1);
             DispValue ('   HP=', 'R', hp,8,1);
             printfln ('');
          end;
       end;
    end;

    if (enable.Sae_rms) then with _Srms do begin
       liner := lines + 1;
       DispValue ('SAE-RMS ['+B_str(dev,2)+']['+B_str(cha,1)+','+B_str(chb,1)+']','S',Bk,0,0);
       DispValue ('  data=', 'R', data,9,5);
       DispValue ('   tc=', 'R', tc  ,8,1);
       IF (mode) then
          buf := 'RMS'
       ELSE
          buf := 'CORR';
       DispValue (' mode=', 'S', buf, 8, 0);
       printfln ('');
    end;

    if (enable.Sae_mean) then with _Smean do begin
       lines := lines + 1;
       DispValue ('SAE-MEAN['+B_str(dev,2)+']['+B_str(cha,3)+']','S',Bk,0,0);
       DispValue ('  data=', 'R', data,9,5);
       DispValue ('   tc=', 'R', tc  ,8,1);
       printfln ('');
    end;

    if (enable.Sae_Sam) then with _Ssam do begin
       lines := lines + 1;
       DispValue ('SAE-SAM ['+B_str(dev,2)+']     ','S',Bk,0,0);
       DispValue ('ACCTIM=', 'R', ACCTIM,9,5);
       DispValue (' Trig=', 'B', Trig  ,8,0);
       DispValue (' Proc=', 'B', Proc  ,8,0);
       DispValue ('Test=', 'B', Test  ,1,0);
       printfln ('');
    end;

    if (enable.Sae_corr) then with _Scorr do begin
       lines := lines + 1;
       DispValue ('SAE-CORR['+B_str(dev,2)+']['+B_str(chX,1)+','+B_str(chY,1)+']','S',Bk,0,0);
       DispValue ('  Idel.=', 'R', Idelay ,9,3);
       DispValue ('Sdel.=', 'R', Sdelay ,8,3);
       DispValue ('Tfac.=', 'B', Tfactor,8,0);
       buf := HEX(status);
       DispValue ('Stat=', 'S', buf    ,5,0);
       printfln ('');
```



```
    end;

    if (enable.trav) then begin
       lines := lines + 1;
       DispValue ('TRAVER              ','S',Bk,0,0);
       DispValue ('     X=', 'R', _trav[0].posn, 9, 1);
       DispValue ('     Y=', 'R', _trav[1].posn, 8, 1);
       DispValue ('     Z=', 'R', _trav[2].posn, 8, 1);
       DispValue ('    R=', 'R', _trav[3].posn, 5, 1);
       printfln ('');
    end;

    if (enable.probe) then begin
       lines := lines + 1;
       DispValue ('PROBE               ','S',Bk,0,0);
       DispValue (' angle=', 'R', _probe_ang, 9, 1);
    end;

    Enable_view (W[DEVICE].handle);

    Appear_Window    (W[DEVICE]);

    if (enable.sae_sam) then begin
       open_sam_window;
       Disable_view (W[SAM].handle);
       GotoXY       (1,1);
       with _Ssam do begin
         printfln  ( S_str ('SAE-SAM ['+B_str(dev,2)+']','C',W[SAM].Xview+1,$3F0) );
         DispValue ('ACCTIM :', 'R', ACCTIM, 12,4);  printfln ('');
         DispValue ('Real T.:', 'R', RealT,  12,4);  printfln ('');
         DispValue ('Remain.:', 'R', RemaInT,12,4);  printfln ('');
         for i:=0 to 7 do begin
           DispValue ('Data['+B_str(i,1)+']:', 'R', Result[i],12,4);
           printfln ('');
         end;
         buf := HEX(status);
         DispValue ('Status :', 'S', buf,12,0);
       end;
       Enable_view   (W[SAM].handle);
       Appear_Window (W[SAM]);
    end;
  END; { UpdateWindow }

{.pa}

{ - - - - CLEAN INPUT CMD     ROUTINES - - - - - - - - - - - - - - - - - -}
{         GET INPUT           ROUTINES                                    }

{$I get-in.inc}


{ - - - - DO LOOP INTERPRETER ROUTINES - - - - - - - - - - - - - - - - - -}
{         WAIT    INTERPRETER ROUTINES                                    }
{         EXECUTE INTERPRETER ROUTINES                                    }

{$I cli-dve.drv }


{ - - - - ENABLE  INTERPRETER ROUTINES - - - - - - - - - - - - - - - - - -}
{         DEBUG   INTERPRETER ROUTINES                                    }
{         HELP    INTERPRETER ROUTINES                                    }

{$I cli-edh.drv }


{ - - - - cli_store_data_file ROUTINES - - - - - - - - - - - - - - - - - -}
{         cli_file            ROUTINES                                    }

{$I cli-file.drv}
```

```
{.pa}

Procedure cli_Read_Write (com    : dev_cpt;     { read_d, set_d or clear_d }
                          device : ValidCmd_t); { device code              }
  VAR   rw_com   : st10;
        read_flg : boolean;
        i        : byte;
  BEGIN
     Push_Msg (' Proc: cli_Read_Write');

        IF (cmd_num >= 0) then
           delay (300);

        CASE device of
          c_all  : begin
                     cli_Read_Write (com, c_rms);
                     cli_Read_Write (com, c_mean);
                     cli_Read_Write (com, c_cond);
                     cli_Read_Write (com, s_rms);
                     cli_Read_Write (com, s_mean);
                     cli_Read_Write (com, s_corr);
                     cli_Read_Write (com, s_sam);
                     cli_Read_Write (com, c_trav);
                     cli_Read_Write (com, c_probe);
                   end;
          c_rms  : CTA_RMS  (com, _Crms.dev,
                                  _Crms.cha, _Crms.tc, _Crms.CL, _Crms.data);
          c_mean : CTA_MEAN (com, _Cmean.dev,
                                  _Cmean.cha, _Cmean.tc, _Cmean.filter,
                                  _Cmean.CL,  _Cmean.data);
          c_cond : FOR i:=0 to 2 DO
                     SIG_COND (com, _Cond.dev, i,
                                    _Cond.cha[i].gain,
                                    _Cond.cha[i].lp,
                                    _Cond.cha[i].hp);
          s_rms  : SAE_RMS  (com, _Srms.dev,
                                  _Srms.cha, _Srms.chb, _Srms.tc,
                                  _Srms.mode, _Srms.data);
          s_mean : SAE_MEAN (com, _Smean.dev,
                                  _Smean.cha, _Smean.tc, _Smean.data);
          s_corr : with _Scorr do begin
                     SAE_CORR (com, dev, chA, chY, Idelay, Sdelay, Tfactor,
                               Status, err_code);
                   end;
          s_sam  : with _Ssam do begin
                     SAE_SAM  (com, dev, Trig, Proc, Test, ACCTIM,
                               RemainT, RealT, Result, Status);
                   end;
          c_trav : IF (com = read_d) then begin
                     TRAVER (com, 0, _trav[0].posn);
                     TRAVER (com, 1, _trav[1].posn);
                     TRAVER (com, 2, _trav[2].posn);
                   end;
          c_probe: PROBE (dev_opt_char[integer(com)], _probe_ang);
        end;
        IF (com in [set_d, clear_d]) and
           (device in [c_all, s_rms, s_mean, s_sam, s_corr,
                       c_rms, c_mean, c_cond, c_trav, c_probe]) then
             UpdateWindow;
     Pop_Msg;
  END; { cli_Read_Write }

Procedure Exit_to_CLI;
  VAR   i : integer;
  BEGIN
     Dispose_All_Loop;

     FOR i:=0 to cmd_num do begin
        CLOSE (cmd_file[i]);
        cmd_file_exist[i] := FALSE;
```



```
        end;
        cmd_num := -1;

        FOR i:=0 to 9 do begin
           IF (file_rec[i].open) then
              CLOSE (data_file[i]);
        end;
  END;

{.pa}

function ConvCmd (Var token3: st3; Var CliCmd: ValidCmd_t): boolean;
  var i : byte;
  begin
    i := 0;
    while ( (i <= max_no_cmd) and (ValidCmd_s[i] <> token3)) do i:=i+1;
    if (i <= max_no_cmd) then begin
      ConvCmd := true;
      CliCmd := ValidCmd_t(i);
    end
    else begin
      ConvCmd := false;
    end;
  end;

Procedure cli_TOKEN (token   : ValidCmd_t;
                     Var buf : st128);
  VAR  dev       : ValidCmd_t;
       com       : st10;
       com3      : st3;
       ch        : char;
       cmd_found,
       tmp       : boolean;
  BEGIN
     Push_Msg (' Proc: cli_TOKEN');

       IF (token = c_do) then
          cli_Begin_DO (buf)

       ELSE IF (token = c_end) then
          cli_End_DO (buf)

       ELSE begin
          REPEAT
             IF ( token in [c_read, c_help, c_enable, c_disable] ) then
                cmd_found := FoundData (buf, com)

             ELSE IF (token = c_wait) then begin
                IF (buf[1] = 'K') then
                   cmd_found := FoundData( buf, com)
                ELSE
                   cmd_found := FoundCom( buf, '=', com);
             end
             ELSE
                cmd_found :=  FoundCom (buf, '=', com);

             IF ( cmd_found ) then begin
                com3 := com;
                case token of
                  c_rms    : cli_CRMS         (com3,  buf);
                  c_mean   : cli_CMEAN        (com3,  buf);
                  c_cond   : cli_COND         (com3,  buf);
                  s_rms    : cli_SRMS         (com3,  buf);
                  s_mean   : cli_SMEAN        (com3,  buf);
                  s_corr   : cli_Scorr        (com3,  buf);
                  s_sam    : cli_Sam          (com3,  buf);
                  c_trav   : cli_TRAV         (com3,  buf);
                  c_probe  : cli_PROBE        (com3,  buf);
                  c_read   : begin
                               if (ConvCmd (com3,  dev)) then
```

```
                            cli_READ_WRITE (read_d,dev)
                          else displayError (4);
                        end;
                c_file   : cli_FILE          (com3,  buf);
                c_wait   : cli_WAIT          (com3,  buf);
                c_execute: cli_EXECUTE_1     (com3,  buf);
                c_enable : begin
                             if (ConvCmd (com3,  dev)) then
                               cli_ENABLE      (dev,  TRUE)
                             else displayError (4);
                           end;
                c_disable: begin
                             if (ConvCmd (com3,  dev)) then
                               cli_ENABLE      (dev,  FALSE)
                             else displayError (4);
                           end;
                c_debug  : cli_DEBUG         (com3,  buf);
              end;
            end;
            IF (KEYPRESSED) then begin
               read (KBD, ch);
               IF (ch = ^[) then begin
                  UpdateWindow;
                  Exit_to_CLI;
               end;
            end;
          UNTIL (not cmd_found);
        end;

        IF (token = c_read) then
           UpdateWindow
        else IF ( token in [c_rms, c_mean, c_cond, s_rms, s_mean, s_corr, s_sam,
                            c_trav, c_probe] ) then
          cli_Read_Write (set_d, token);

    Pop_Msg;
  END; { cli_TOKEN }

{.pa}

Procedure MAIN;
  VAR  buf            : st128;
       command        : st10;
       com_found      : boolean;
       com3, token3   : st3;
       CliCmd         : ValidCmd_t;
       ch             : char;
       i              : byte;
       dos_param_flag : boolean;
       dispHelp       : boolean;

  BEGIN
    if (ParamCount > 0) then begin
      buf := '';
      dos_param_flag := true;
      for i:=1 to ParamCount do begin
        buf := buf + ParamStr(i) + ' ';
      end;
      clean_input_cmd (buf);
    end

    else begin
      dos_param_flag := false;
    end;

    token3 := '';
    REPEAT
      Open_cmd_window;

      if (dos_param_flag) then begin
```



```
        dos_param_flag := false;
        printfln ('');
        printf   ('Init CMD > ' + buf);
      end

      else IF (NOT _do_flg) then begin
        Get_Input ('', buf);
        token3 := buf;

        IF (token3 = 'DO:') then begin
          if (cli_before_DO (buf)) then begin
            buf := _do_rec_ptr^.text;
            _do_rec_ptr := _do_rec_ptr^.next;
          end
          else begin
            _do_flg := false;
            buf := ';';
          end;
        end;
      end

      ELSE begin
        buf := _do_rec_ptr^.text;
        _do_rec_ptr := _do_rec_ptr^.next;
        printfln ('');
        write ('Loop:', _do_cur_lab:1, ';',
                        _do[_do_level].ctr:3, ' > ', buf);
      end;
      Update_Time_Date (true);
      Open_user_window;

      token3 := buf;
      dispHelp := false;

      if (buf[1] = ';') then
        begin end                                    { skip line "command'}

      else if ( ConvCmd (token3, CliCmd) ) then begin
        (* A table of valid commands are defined in globle area *)

        Open_user_window;

        if ( CliCmd = c_dos ) then begin
          Ram_page       (0);                        { open system window }
          Appear_View    (0);
          TextColor      (white);
          TextBackground (black);
          Enable_cursor;
          ClrScr;
          delete  (buf, length(buf), 1);
          cli_dos (buf);
          ClrScr;
          Display_All_Windows;
        end.

        else if (FoundCom (buf, ':', command)) THEN BEGIN
          IF ( CliCmd in [c_rms,    c_mean,    c_cond,  s_rms,    s_mean,
                          s_sam,    s_corr,    c_trav,  c_probe,  c_lda,
                          c_debug,  c_disable, c_do,    c_enable, c_end,
                          c_file,   c_execute, c_read,  c_wait] )
          then begin
            cli_TOKEN (CliCmd, buf);
          end
          else dispHelp := true;
        end

        else IF (CliCmd = c_help) then
          cli_help

        else if (CliCmd = c_window) then begin
```

```
          Change_Screen (User, All);
          Display_All_Windows;
        end

        else if ( CliCmd in [c_exit, c_quit] ) then begin
          Exit_to_CLI;
        end

        else dispHelp := true;
      end

      else dispHelp := true;

      if (dispHelp) then begin
        open_cmd_window;
        printfln ('');
        printfln ('* Invalid Command'+ Bell);
        printf   ('* type HELP for more info.');
      end;

      IF (KEYPRESSED) then begin
        read (KBD, ch);
        IF (ch = ^[) then
          Exit_to_CLI;
      end;
    UNTIL (CliCmd = c_quit);
  END; { MAIN }

{.pa}

Procedure Init_Sys_Flgs;
  VAR i : byte;
  BEGIN
    debug_488      := 0;              { no debug message will be disp.}

    _help_loaded   := false;

    _do_flg        := FALSE;
    _do[0].first   := nil;

    FOR i:=0 to 9 do
       file_rec[i].open := FALSE;

    cmd_num := -1;
    FOR i:=0 to 10 do
       cmd_file_exist[i] := FALSE;

    Bell := #15;
    DOS_mem := 0;

    SetCrtMode (c80);
    Enable_cursor;
    Open_Windows;

    if ( Init_GPIB ('CTA', cta_id, err_msg) ) then
      printfln ('cannot init GPIB-CTA : ' + err_msg);

    sae_id := cta_id;
    Init_cta_drv;        Init_cli_cta;
    Init_sae_drv;        Init_cli_sae;       Init_cli_sam;

    if ( Init_GPIB ('TRAV', trav_id, err_msg) ) then
      printfln ('cannot init GPIB-TRAV : ' + err_msg);

    Init_trav_drv;       Init_cli_trav;

    Init_IOtech;         Init_cli_probe;

    DOS_mem := MemAvail / 64.0;
    Init_dos (memAvail - (10 * 1024 div 64));              { paragraph - 10k }
```



```
    DOS_mem := DOS_mem - (MemAvail / 64.0);

  END; { Init_Sys_Flgs }

{.pa}

VAR result             : integer;
    r,c,cm             : byte;
    err_msg            : st128;

begin
  Init_Sys_Flgs;
  Init_msg_stack;

  Open_CMD_Window;

  printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
  printfln ('                Welcome to CADA-LDA             ');
  printfln ('                                                ');
  printfln ('       Computer Aided Data Acquisition System'); 
  printfln ('              for the CTA environment           ');
  printfln ('              Written by :  Alex Taui           ');
  printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');

  delay ( 1500 );

  if true then begin
    for i:=0 to 15 do printfln ('');

    printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
    printfln ('       Login TIME & DATE > ' + get_time + ' ' + get_date);
    printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
    printfln ('       Computer Aided Data Acquisition System');
    printfln ('       for :  Constant Temperature Anemometer');
    printfln ('              Signal Analysis Equipements     ');
    printf   ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');

    Main;
    SetCrtMode (c80);
  end;

END.
```

CADA-CTA.DEF

```
User Window
        79        25
         2        12        55        14
        53         2        26        23
         0         2
         1        22
       255
Input Window
        79        25
         2        16        55        24
        53         8        26        17
         0         3
         1        17
       255
Device Window
        85        25
         2         3        79        10
        77         7         8        18
        15         1
         1         1
       255
SAE-SAM Window
        28        15
        58        12        79        24
        21        12         7         3
        15         1
         1         1
       255
Help Window
        79       450
         2         4        79        20
        77        16         2       419
        15         0
         1         1
       255
Time & Date Window
        80         1
         1         1        80         1
        79         0         1         1
        15         4
         1         1
         0
Dummy Window
         2         2
         1         1         2         1
         1         0         1         2
        15         4
         1         1
         0
```


MAKE.BAT

```
echo off

if "%1" == "" goto Shellgen
  tm start
  goto Compile

:Shellgen
  md  g:\tmp
  del g:\tmp

  tm start
  shellgen -B  -R -S -O g:\tmp cada-cta
  tm stop

:Compile
  copy bigturbo.var       g:\tmp
  copy farctrl.inc        g:\tmp
  copy loadmod.inc        g:\tmp

  copy ram-page.bin       g:\tmp
  copy spawn4ah.bin       g:\tmp
  copy spawn4bh.bin       g:\tmp
  copy dd-ni488.bin       g:\tmp

  g:
  cd \tmp
  bigmake -A -C -N -O f: cada-cta
  tm stop
  f:
```

APPENDIX C - COMMAND DESCRIPTION FOR CADA-LDA

Command Syntax

SYNTAX: CADA-LDA <CADA-LDA command line>

The syntax used to invoke this program in the DOS environment is shown above.

An optional command line may be included during the invocation of this program. This optional command line will be treated as the first line to be interpreted by this software.

If this option is not included, this software will wait for the first input after it is initialized.

example: To invoke CADA-LDA software and let the software prompt for input command.

```
type >> CADA-LDA
```

The following syntax can be used to invoke the LDA program and execute a command file to do automatic data acquisition.
(The user must write the command file "acquire.cmd" with an editor before typing in the following line)

```
type >> CADA-LDA execute:file=acquire.cmd;
```

Definition of this help file

```
Primary_Cmd : Sub_Cmd = Value [, Sub_Cmd = Value] ;
   ***      *   ***   *  ***   *   ***   *  ***   *
            ^       ^      ^       ^       ^      ^
```

*** -> Primary or Secondary/Option command.
* -> Separators.

NOTE: Don't forget the separators. They are essential !!

All primary commands must end with a colon ---------> ' : '.
The secondary command separator is a comma ---------> ' , '.
All command lines must terminate with a semi-colon -> ' ; '.
All string after ';' will be considered as comments.

n2= Integer values in the range of 0 to 2 (ie 0,1, .. 2)
n7= Integer values in the range of 0 to 7 (ie 0,1, .. 7)
r = Real number (ie 1.128)
Ir= Incremental value (ié +0.25 or -1.115)

Note: 1) Tokens within a set of square brackets are used to indicate that none or one of the specified options can be selected.

ie: If [token1 or token2 or token3] are given, none or one can be chosen from that list.

2) Only the first three characters of the token or command are important.

3) Blanks are neither token separators nor of any importance except when a DOS command is made.

Note: <ESC> can be used to terminate an executing command or command file.

DEVICE control commands

[Primary] [Secondary Token]
ENABLE: LDA, TRAVERSE, AINPUT, AOUTPUT, ALL ;
DISABLE: LDA, TRAVERSE, AINPUT, AOUTPUT, ALL ;
READ: LDA, TRAVERSE, AINPUT, AOUTPUT, ALL ;

example: To select the LDA and TRAVER units, the follow command must be given.

CMD > enable: lda, traver;

The follow command will read all the device(s) which one have ENABLEd.

CMD > read: all;

or to read just one of the enabled device.

CMD > read: lda;

If one wishes to deselect a device, the DISABLE command should be used.

CMD > disable: lda;

LDA Counter Processor programming command

[Primary] [Secondary Token and default setting]
LDA: Mode = 0,
Nsets = 0,
Interval time = 0,
Display Mode = ON,
DMA channel # = 3,
Time Base = 2,
Delay Factor = 1.00;

[Description of the Secondary Token (Sub-command)]
Mode = n4, - All modes produce Sample Interval time.
mode 0 : Raw data of P, F, T and D/
mode 1 : Fixed mode yields Doppler frequency.
mode 2 : Combined mode yields Doppler frequency and fringe count.
mode 3 : Burst mode yields burst time and fringe count.
mode 4 : Transit mode yields transit time.

Nsets = n10000, - Number of consecutive readings.
(accept range from 0 to 10,000).

Interval time = r, - Approximate time delay between each consecutive reading.
(accept range from 0.05 sec to 32.0 sec).
If it is set to 0.0, the DMA mode will be activated.
This mode may acquire data as fast as 500K byte/sec or approximately 83 thousand sets of data per second which depend on the signal data rate.

Display Mode = [ON or OFF or AGAIN]
- ON : Display results in the data window after a "READ : LDA" command is send.
- OFF : Suspend data display after reading.
- AGAIN : Re-display the LDA results on the data window.

DMA channel# = [1 or 3],
- depends on PC hardware configuration.

TimeBase = n3, - This selects the time base for the sample interval calculation.
0 : time base = 1. (external time base used).
1 : time base = 0. (not allowed).
2 : time base = 1e-3 sec.
3 : time base = 100e-9 sec.

Delay Factor = r, - This factor is used to generate the actual delay between each consecutive reading when the Interval time is set to non-zero.
- The default setting is 1.00. This value is good for running on the standard 4.77 Mhz. IBM-PC, and may be required to be changed if other compatable systems are used.

example: To program the CADA-LDA software to receive 20 sets of readings as fast as the LDA counter processor can generate at mode 1 (FIXED MODE), and set the Sampling Interval Time Base to 1 msec. One should type in:

```
CMD > LDA : Nsets = 20, Interval_Time = 0.0, Time Base = 2;
CMD > READ: LDA;
```

The "READ : LDA" command will read the data from the LDA counter processor.

If one wishes to reprogram a particular setting, one must type in a command similar to the one above with the new information. For example, if one wants to change the sampling rate to 5 readings per second, the following must be entered.

```
CMD > lda : Interval time = 0.2;
CMD > read: lda;
```

To re-display the data, just read. One must type in:

```
CMD > lda : display = again;
```

example: If one is using an IBM-AT to run this software, it may be required to change the DELAY FACTOR to generate a desirable delay between consecutive readings (w.r.t a particular INTERVAL TIME setting). To determine such delay factor, one may follow this example.

1 - assume the interested INTERVAL TIME is 0.1 sec.
2 - configure the LDA to read 600 sets of readings, with DELAY FACTOR set to 1.0.
(note: if the DELAY FACTOR is correct, this should take 60 sec. (theoretical time))
3 - Measure the actual time for to take this 600 sets of readings.
4 - The new DELAY FACTOR for this particular example will be DELAY FACTOR = (theoretical time) / (actual time).

```
CMD > lda : Nsets = 600, Interval Time = 0.1, Delay Factor = 1.0;
CMD > ;
CMD > read: lda; ... start stop watch ...
CMD > ;
CMD > ;          ... new DF = theoretical time / actual time ...
CMD > ;
CMD > lda : Delay Factor = <new DF>;
```

TRAVERSING DEVICE programming command

```
[Primary] [Secondary Token]
TRAVERSE: =INIT ;                              - init. posn. ctr. to 0.0
TRAVERSE: X = [r, Ir], Y=[], Z=[], R=[],       - Ir => increment by r mm.
          DEVICE_NUM = n15,                    - device num. for the X axis.
          DELAY= [r or WAIT or ALL_FINISH] ;   - delay before the next trav.
                                                 command will be send.
```

example: The TRAVERSE: command is designed to initialize and control the positioning of the axes which connect to the TRAVERSING unit.

The INIT sub-command is used to zero the position counters. After the traverse axes are positioned to a known datum point in a particular experimental setup, issue the following:

```
CMD > traverse : =init;
```

The X axis is addressed to the device which specified by the DEVICE_NUM parameter, where axis Y is one higher then DEVICE_NUM, and Z is two higher then DEVICE_NUM.

To program the traverse axes to location X = 10.5mm, Y = 2.125mm and Z = 1.0mm from a datum position and wait for all three axis movements to stabilize before executing the next command, the following should be enter.

```
CMD > traverse : x=10.5, y=2.125, z=1, delay=all_finish;
```

if the DELAY=WAIT option is used, the software will only wait for the last specified axis (namely the Z axis).

To increment the traverse from the current X location to a new position 100mm in the negative direction, one should issue an incremental offset as the new location.

```
CMD > traverse : x=i-100.0, delay=5.5;
```

The delay option specified in the above command has signified a 5.5 second delay before interpreting the next input command.

If more then one traverse unit is connected to the system, one can issue the following command to select the second traversing unit. Assume the device num (DCBA switch setting) for the first traversing unit is 0, the second traversing unit is set to 8, and both of the units are initialized correctly.

```
CMD > traverse: device_num = 8; select device 8 (2nd trav. unit)
CMD > read:   traverse;         update software posn. counters
CMD > traverse: x=100, ..... ;  program second traversing unit.
..
..
CMD > traverse: device_num = 0; deselect device 8 and select 0
CMD > read:   traverse;         update software posn. counters.
```

ANALOG INPUT/OUTPUT programming command

Analog Input command:

```
[Primary] [Secondary Token]
Ainput:   LOW  CHANNEL = n15,                - First channel.
          HIGH CHANNEL = n15,                - Last  channel.
          COUNT        = n32767,             - Number of consecutive readings.
          RATE         = n32767,             - Number of samples per second.
          OPTION       = <string>,           - Refer to ISAAC reference menu.
          DISPLAY      = [ON | OFF | AGAIN]; - Display after read ?
```

[Description]
This command is designed to scan a sequence of channels (twelve-bit Analog to Digital Converter channel) in ascending numerical order. LOW and HIGH identifies the channel sequence. COUNT define how many sets of reading to take, RATE define how fast to sample each set of data, while an OPTION command is available to change the ISAAC default setting, to your needs. The DISPLAY command controls how the result is displayed after a read operation.

```
LOW    = n15,          - Indicates the starting channel.
HIGH   = n15,          - Identifies the ending channel.
                         If LOW and HIGH are equal, only one
                         channel is scanned.

COUNT  = n32767,       - This sub-command indicates how many times to scan the
                         channels. The total number of samples
                         input is the number of channels scanned times the
                         count.

RATE   = n32767,       - This is used to select the scanning rate. This para-
                         meter controls the rate of which each sequence is
                         sampled.  Within each set of channels the inter-channel
                         time is as short as possible; RATE determines the rate
                         at which complete scans are executed. If rate is equal
                         to zero, external clocking will be used. Reference the
                         ISAAC documentation for information.

OPTIONS = <string>,    - The valid optional parameters are the standard DEVICE,
                         UNIT, and CONTROL parameters, the EXECUTION mode para-
                         meter, the STORAGE parameter, and the INTSOFF flag.
                         Refer to page 4-2 in the ISAAC documentation for
                         detail information.

Display Mode = [ON or OFF or AGAIN];
                    - ON    : Display results in the data window after
                              a "READ : LDA" command is send.
                    - OFF   : Suspend data display after reading.
                    - AGAIN : Re-display the LDA results on the data window.
```

Analog Output command:

```
[Primary] [Secondary Token]
Aoutput:  CHANNEL #  = n15,          - Analog output channel.
          DAC        = n4095,        - Digit to Analog value.
          OPTION     = <string>,     - Refer to ISAAC reference menu.
```

[Description]
This command takes a digital value and outputs it to the specified twelve-bit digital to analog converter channel.

```
CHANNEL # = n15,       - This selects the analog output channel.

DAC       = n4095,     - This is the digital value the ISAAC system used to
                         convert to an analog voltage. a zero implies -5 volts.
                         while DAC equal 4095 implies a +5 volts.

OPTIONS = <string>,    - The valid optional parameters are the standard DEVICE,
                         UNIT, and CONTROL parameters, the EXECUTION mode para-
                         meter, the STORAGE parameter, and the INTSOFF flag.
                         Refer to page 4-2 in the ISAAC documentation for
                         detail information.
```

example: To scan 10 sets of analog input data starting at channel 2, and end at channel 5, issue the following.

```
ex:  CMD > Ainput: low = 2, high = 5, count = 10;
```

To set analog output channel 3 to 3.5 volts, first the 3.5 volts constant must be converted to a digital representation. the following are required:

```
since      0 = -5 volts, and 4095 = +5 volts.
therefore 3.5 volts = ((3.5 + 5)/10) * 4096
                    = 3481.6
                    - 3482

ex:  CMD > Aoutput: channel = 3, dac = 3482;
```

DOS LEVEL COMMUNICATION COMMANDS

```
[Primary] [Secondary Token]
DOS [-p] ;
```

- Open a subprocess to DOS, type EXIT to return to CADA-LDA program.
- If the '-p' option is given, CADA-LDA will bypass the question 'press RETURN to continue....'

```
ex : CMD > DOS;
     C> setpath                    : execute a bat program
        .. .. .. ..
     C> dir/w                      : display directory
        .. .. .. ..
     C> lotus                      : run a DOS program
        .. .. .. ..
     C> exit                       : return to CADA-LDA program

                        press RETURN to continue ....
```

```
DOS [-p] := <d:\path\filename [param.]> ;
```

- Open a DOS process and execute the given DOS command.
- The DOS command can be a DOS SHELL, f.COM, f.EXE or f.BAT .
- If the '-p' option is given, CADA-LDA will bypass the question 'press RETURN to continue....'

```
ex : CMD > DOS -p := lotus;
```

After the above line is entered, CADA-LDA will open a subprocess to execute lotus. At this point, one can edit a work file, etc. But, upon exiting the lotus program, the user will find that he/she is back in the CADA-LDA program.

SYSTEM and MISC. commands

```
HELP ;    (or F1)
```

- will display this file.
- one can control the movement of the window with the cursor keys (namely : HOME, END, Up, Down, Left, Right, PgUp and PgDn)
- press F10 to exit the help window.

```
QUIT ;    - This will close all opened files and return to DOS.

WINDOW ;  (or F2)
```

- Allow the user to use CURSOR KEY to change window setup.
  - F5 - Scroll screen.
  - F6 - Size screen.
  - F7 - Move screen.
  - F8 - Change Window (this is done in a circular manner)
  - F9 - Save, Recall or Delete window configuration file.
  - F10- EXIT window setup menu.

LOOPING and WAIT commands

```
[Primary] [Label] [Secondary Token]
DO:       n9,     TIMES=n32767 ;        - The label of each DO command
 ..                                       must match with an END
END:      n9 ;                            command of the same label.

WAIT:     KEY,                          - wait for key pressed
          TIME=hh:mm:ss,                - wait until hh:mm:ss
          DELAY=r ;                     - delay for r second
```

example: To take 100 readings from the LDA counter processor unit starting at 6:30 pm. from XYZ = (0, 0, 0) in every 1mm in the X direction with a 100 second delay for the RMS reading to be stabilized.

```
CMD > lda      : nsets=5, interval=0.1;        configure the LDA.
CMD > wait     : time=18:30:00;
CMD > traverse: x=0, y=0, z=0, delay=all_finish;
CMD > do: 1, times=100;
CMD >     wait: delay=100.0;
CMD >     read: lda;
CMD > end: 1;
```

CMD file EXECUTION commands

```
[Primary] [Secondary Token]
EXECUTE:  FILE=<drive:path\filename.ext> ;  - execute a sequence of
                                              programmed command.
```

example: One can write his own controlling command file with his choice of editor and execute them within the CADA-LDA program. For example, if the command of the last example (exclude 'CMD >') is stored in a file called READ-LDA.cmd, one can invoke it by using the EXECUTE command in the following way.

```
    CMD > execute: file= read-lda.cmd;
```

If this cmd file is located in a different directory, one should include a path name.

```
    CMD > execute: file= \lda-dir\read-lda.cmd;
```

FILE handling command

```
[Primary] [Secondary Token]
FILE:     [   OPEN  = <filename.ext>          - open a data file
           or CLOSE                           - close a data file
           or FORMAT=( .. )                   - define write format
           or STORE ],                        - store data to file
          TO_FILE_# = n9 ;                    - to file number n9


          where ( .. ) = ( TIME, DATE,        - generate time/date
                           SP=n,              - generate n space(s)
                           AINPUT,            - generate analog input data
                           AOUTPUT,           - generate analog output data
                           LDA,               - generate LDA data
                           TRAVERSE )         - x,y,z posn. in mm
```

example: A particular experiment requires 50 sets of data to be read from the LDA counter processor approximate every 2.5 min. Each set of data consist of 10 consecutive readings in 0.1 second apart. This also requires the LDA unit to be running at mode 1 (acquiring the Doppler frequency and the Sample interval time). The result of these readings are to be stored to a text file called 'LDA.dat'. In addition, the full status of LDA readings are to be printed to the printer as the experiment progresses. Then a FORTRAN program (called ANALYSIS.exe) is used to analysis the set of data and print the result to a printer.

```
    CMD > enable : lda, traverse;
    CMD > lda : nsets = 10, interval = 0.1;
    CMD > ;
    CMD > file: open   = LDA.dat, to_file_# = 1;
    CMD > file: format = (time, LDA), to_file_# = 1;
    CMD > ;
    CMD > file: open   = prn, to_file_# = 9;
    CMD > file: format = (time, LDA), to_file_# = 9;
    CMD > ;
    CMD > do : 1, times = 50;
    CMD >     wait : delay = 150;          delay for 2.5 min (150s)
    CMD >     read : LDA;
    CMD >     file : store, to_file_# = 1;
    CMD >     file : store, to_file_# = 9;
    CMD >     ;
    CMD > end: 1;
    CMD > ;
    CMD > file: close, to_file_# = 1;
    CMD > file: close, to_file_# = 9;
    CMD > ;
    CMD > dos := ANALYSIS <LDA.dat >prn;
```

.. END OF HELP ..

APPENDIX D - SOURCE LISTING OF THE CADA-LDA PACKAGE

README.LDA

A: SYSTEM REQUIREMENTS

To use this package, you must have the following:

1: An IBM-PC, an IBM-XT or an IBM-AT (or a compatible).

2: The CADA-LDA package requires approximately 384k of RAM memory.

3: One 360k disk drive is sufficient.

4: DISA Laser Doppler Anemometer Counter Processor.

5: PC-LDA interface card.

6: DISA four Axes Traversing Unit.

7: Cyborg ISAAC 91A Analog I/O Subsystem.

4: The NATIONAL INSTRUMENTS "GPIBPC2A" interface are required.

Switch settings are as follows: BASE I/O addr :: _____
DMA channel :: _____
Interrupt line:: _____

B: TO START THE CADA-LDA PACKAGE (from floppy drive)

1: Insert the supplied diskette into the BOOT drive. (A:).

2: Turn the computer 'ON'.
This supplied diskette contains the necessary DRIVERs required by this package to run correctly if the above hardware requirements are met.

3: The AUTOEXEC batch will bring up the CADA-LDA software.

C: Installing the CADA-LDA package to the hard disk drive

1: Boot system.

2: Insert the supplied diskette into drive A:.

3: Make a directory in the hard drive with the name CADA, change directory to CADA and copy all files from A: to CADA.

```
C:> md \CADA-LDA
C:> cd \CADA-LDA
C:> copy a:*.* .
```

D: Execute the CADA-LDA program from the hard disk system

1: Type in the following :

```
C:> cd \CADA-LDA
C:> CADA-LDA
```

C: FILES SUPPLIED

This entire package is supplied on a single diskette which contains the executable files as well as some example command files. You will find the following files on the diskette supplied.

AUTOEXEC.BAT	- Batch file used to invoke the CADA-LDA package.
CONFIG .SYS	- This file must be present when BOOTING the system.
CADA-LDA.EXE CADA-L .COM	- Executable files for the CADA-LDA package.
CADA-LDA.DEF	- Window definition file is used internally to maintain windows information.
CADA-LDA.HLP	- On-Line help file.
GPIE .COM	- Communication driver for the NATIONAL INSTRUMENTS GPIB-PC2A interface.
README .LDA	- Contains this file.
* .CMD	- Example command file which can be executed by the CADA-LDA package.

FILES REQUIRED TO DO CADA DEVELOPMENT WORK

Text files :

readme	lda	: Include - Instruction for software installation - Source files description

Files required to run CADA-LDA program

cada-lda	exe	: Start-up and ISAAC communication module
cada-l	com	: Window management module
cada-lda	hlp	: On-Line help file
cada-lda	def	: Window definition file

Sample command files for CADA-LDA program

Example command files :

isaac	cmd
lda0	cmd
lda1	cmd
lda2	cmd

Example data file :

isaac	dat

Programs needed to do CADA development work

Compiler and other programs

TDEBUG	COM	: Turbo Pascal debuger
TURBO	MSG	: Turbo Pascal compiler message file
BIGMAKE	COM	: Make utility for turbo pascal
BTURBO	COM	: Special version of turbo which supports the make utilities.
SHELLGEN	COM	: Generate turbo program bigger than 64K
BIGTURBO	VAR	: files required by SHELLGEN, BIGMAKE & BTURBO.
FARCTRL	INC	
LOADMOD	INC	

Intstruction to compile the CADA-LDA package

1: compile the CADA-LDA.C program

```
link /M/L \lc\l\c+cada-lda.c ,,/n,\lc\l\lc+isaac\labcl.lib
```

2: compile the CADA-L.PAS program, this process required a make file call 'cada-l.mak'.

```
bigmake cada-l
```

Files required to create CADA-LDA.EXE
=====================================

cli	bat	: Batch file used to create CADA-LDA.EXE
make	bat	: Batch / Make files for compiling CADA-L.COM
cada-l	mak	
td	bat	: Batch file used to debug CADA-L.COM

Source files for the start-up and ISAAC communication module (CADA-LDA.EXE)
==

This C program starts up the data acquisition environment and allows the CADA-L.COM program to communicate to the ISAAC hardware
---

cada-lda	c	

Assembler program used to control the ISAAC hardware
---

si_drv	asm	

Source files for the window management module (CADA-L.COM)
==

Binary files used with the following source files
---

dd-ni488	bin	: Device driver for National Instrument's IEEE-488 card
ram-page	bin	: Driver for the windowing interface
spawn4ah	bin	: DOS Call interface
spawn4bh	bin	: DOS Call interface

Pascal source code for CADA-L.COM
=================================

Main program :
---

cada-l	pas	: Command interpreter and window management module

Command Interpretation modules :
---

cli-dos	drv	: Implementation of 'DOS' command
cli-dwe	drv	: Implementation of 'DO', 'Wait' & 'Execute' command
cli-help	drv	: Implementation of 'Help' command
cli-lda	drv	: Implementation of 'LDA' command
cli-tra	drv	: Implementation of 'TRAVERSE' command

Device driver modules :
---

dd-gpib	drv	: Interface module for the GPIB / IEEE-488 protocol
dd-isaac	drv	: Interface module for the ISAAC Analog I/O control
dd-ni488	drv	: National Instruments IEEE-488 interfacing code
dd-lda	drv	: Low level control for the LDA counter processor
dd-trav	drv	: Convert high level command to IEEE-488 calls

Utility modules :
---

spawn4ah	asm	: DOS CALL interface
spawn4bh	asm	: DOS CALL interface
er-stack	inc	: General debugging routines
get-in	inc	: Keyboard routines
library	inc	: General utilities routines
ram-page	inc	: Windowing interface routines
rwindow	inc	: Windowing interface routines
spawn	inc	: DOS program spawning routines

string	inc	: String utilities

```
Directory of \CADA-LDA

si_dvr   asm      4,960   1-27-87   9:41p
spawn4ah asm      2,304  11-13-86  11:06a
spawn4bh asm      4,992  11-13-86  11:07a
cli      bat         73   2-22-87  11:26p
make     bat         58  10-12-88  10:59p
td       bat         88   1-27-87  11:35p
dd-ni488 bin        343   7-14-86   3:35p
ram-page bin      5,700   4-09-86  11:13a
spawn4ah bin         58  11-13-86  11:12a
spawn4bh bin        111  11-13-86  11:12a
cada-lda c        8,240  11-11-88   4:27p
isaac    cmd        448   1-29-87  12:18a
lda0     cmd        292   2-26-87   9:26a
lda1     cmd      1,271   2-26-87   9:26a
lda2     cmd        322   2-26-87   9:25a
cada-l   com     59,561  10-13-88   4:19p
isaac    dat        603   2-26-87   9:21a
cada-lda def      1,059  10-12-88  11:13p
cli-dos  drv      2,763   1-20-87  12:52p
cli-dve  drv      8,786  11-11-88   4:20p
cli-help drv      1,088  10-06-88  12:41a
cli-lda  drv      8,453  11-11-88   4:21p
cli-tra  drv      5,589   1-25-87   8:11p
dd-gpib  drv      6,995  11-09-86   4:33p
dd-isaac drv     11,645  11-11-88   4:22p
dd-lda   drv      8,500  11-11-88   4:23p
dd-ni488 drv      7,463  10-12-88  11:07p
dd-trav  drv      5,221  11-27-86  12:32p
cada-lda exe     24,128   2-22-87  11:49p
cada-lda hlp     28,086  10-31-88   4:31p
er-stack inc      4,447  11-07-86  12:19a
get-in   inc      3,732  11-11-88   4:23p
library  inc      4,490  11-25-86  11:41a
ram-page inc      1,399  10-04-88  10:22p
rwindow  inc     15,732  11-11-88   4:24p
spawn    inc      8,011  10-12-88  11:10p
string   inc      2,909  11-11-88   4:24p
readme   lda      7,033  10-31-88   4:35p
cada-l   mak        409  10-13-88   4:19p
cada-lda obj      4,926   2-22-87  11:49p
si_dvr   obj        416   1-27-87   9:42p
cada-l   pas     36,008  11-11-88   4:26p

  45 files found    1,257,472 bytes free
```



```
/* ------------------------------------------------------- */
/*  Written by  : Alex Tsui                                */
/*  Date        : 23:44:58  1/22/1987                      */
/*  Last Update : 23:11:28  2/22/1987                      */
/* ------------------------------------------------------- */
/*  Title       : Software Interrupt Driver in Lattice C   */
/*  Require     : S_int_as.asm    (asm. language driver)   */
/* ------------------------------------------------------- */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <time.h>

/* ---------- LabSoft Functions ---------------------------- */
extern   short  ainfm(),ainm(),ains(),ainsc(),aints(),aoufm();
extern   short  aoum(),aous(),aousc(),bcdinm(),bcdints(),bcdoum();
extern   short  bcdous(),beepfun(),binm(),bins(),bints(),bitins();
extern   short  bitints(),bitous(),bous(),bous(),cins(),cins();
extern   short  cints(),compat(),contin(),cset(),delay(),dins();
extern   short  dous(),normal(),stats(),stinm(),stins(),stints();
extern   short  stopfun(),tinfm(),tinm(),tins(),tints(),tlin();
extern   short  version();

#define  max_func_num  43

short    *Function_Table[]
           = { /*  1 */ ainfm,    ainm,    ains,    ainsc,   aints,
               /*  6 */ aoufm,    aoum,    aous,    aousc,   bcdinm,
               /* 11 */ bcdints,  bcdoum,  bcdous,  beepfun, binm,
               /* 16 */ bins,     bints,   bitins,  bitints, bitous,
               /* 21 */ bous,     bous,    cinm,    cins,    cints,
               /* 26 */ compat,   contin,  cset,    delay,   dins,
               /* 31 */ dous,     normal,  stats,   stinm,   stins,
               /* 36 */ stints,   stopfun, tinfm,   tinm,    tins,
               /* 41 */ tints,    tlin,    version
             };

/* ---------- define the interrupt vector number ----------- */
#define  intr_addr  0x60

/* ---------- declare external program --------------------- */
extern   int  SI_DVR();

/* .ps */

/* ---------- variable used in the interrupt routine -------- */
int      pc_time ();
unsigned char __no_of_ticks,  __StartCol,  __Attr;
union { struct   addr_type  { short ofs;
                              short seg; } addr;

        short    *ptr;
      } data;

/* ---------- declare for DOS interrupt call ---------------- */
union    REGS  inregs, outregs;
struct   SREGS segregs;

/* ----------- structure used to connect an interrupt routine */
struct   SI_type   { long  dummy;
                     short DSeg;
                     int   *old_ptr;
                     short busy;
                     short valid_no_of_func;
                     char  id_length;
                     char  id[13];
                     short param[20];
                     short *func_table[50]; } *SI;

/* ---------- when a 0 is assigned to intr_table, it become
```

```
            an array identifing the interrupt vector table  */
int   **intr_table;

/* ---------- this structure define the address of the color
            text screen location                              */
union {  struct   screen_type      { char  ch;
                                     char  attr; } *p;
         struct   screen_addr_type { short lsb;
                                     short msb;  } addr;
       } screen;

/* .pa */

/* ---------- variable used to store the current time & date  */
long  timeValue;

int init_pc_time ()
{ /* set up screen address ----------------------*/
  screen.addr.lsb       = (short) 0x0000;

  /* read current video state -------------------*/
  inregs.h.ah = 0x0f;
  int86x (0x10, &inregs, &outregs, &segregs);

  if (outregs.h.al == 0x07) {                   /* monochrome monitor  */
    screen.addr.msb = (short) 0xb000;
    __StartCol      = 56;
    __Attr          = 0x70;                     /*  reverse display    */
  }
  else {                                        /* color monitor ------*/
    screen.addr.msb = (short) 0xb800;
    __Attr          = 0x2F;                     /*   B. White on Green */
    if ((outregs.h.al & 0x02) == 0)  __StartCol = 16;
    else                             __StartCol = 56;
  }

  /* initialize the start time ------------------*/
  time (&timeValue);

  /* read date from real time clock */
  inregs.h.ah = 0x00;
  int86x (0x1a, &inregs, &outregs, &segregs);

  timeValue -= ((((outregs.x.cx << 16) + outregs.x.dx) * 10) / 182);
}

/* .pa */

short pc_time ()
{ union  REGS  inregs, outregs;
  struct SREGS segregs;
  char   *timeStr, i, j;
  long   cur_time;

  /* read time from real time clock */
  inregs.h.ah = 0x00;
  int86x (0x1a, &inregs, &outregs, &segregs);

  cur_time =  timeValue;
  cur_time += ((((outregs.x.cx << 16) + outregs.x.dx) * 119318) / 65536);

  timeStr = ctime (&cur_time);

  j = __StartCol;
  for (i=0; i<24; i++) {
    screen.p[j].attr = __Attr;
    screen.p[j].ch   = timeStr[i];
    j++;
  }
  return (0x10);
```



```
}

/* .pa */

char ver[] = "1234";
char rel[] = "12345678";

test_func ()
{ char ch, value;

  value = 0;
  do {
    printf ("\n ... <CR> to continue ...");
    getch ();
    printf ("\x1b[2J Select a command <0-display time, 1-Get ISAAC Ver# or q-Quit> ? ");
    ch = getch ();
    switch (ch) {
      case '0' : { inregs.x.bx = 0;          /* func_number              */
                   inregs.x.cx = 0;          /* num. of parameters passed */
                   int86x (intr_addr, &inregs, &outregs, &segregs);
                   printf ("\n error code = %x", outregs.x.ax);
                   break;
                 }
      case '1' : { inregs.x.bx = 43;         /* get version number       */
                   inregs.x.cx = 4;          /* num. of parameters passed */
                   data.ptr      = ver;
                   SI[0].param[0] = data.addr.ofs;
                   SI[0].param[1] = data.addr.seg;
                   data.ptr      = rel;
                   SI[0].param[2] = data.addr.ofs;
                   SI[0].param[3] = data.addr.seg;
                   int86x (intr_addr, &inregs, &outregs, &segregs);
                   printf ("\n ISAAC Version: %s, Release date: %s", ver, rel);
                   printf ("\n error code = %x", outregs.x.ax);
                   break;
                 }
    }
  } while ( ch != 'q' );
}

/* .pa */

main (argc, argv)
int  argc;
char *argv[];
{ char  i;

  /* define interrupt table pointer */
  intr_table          =  0;

  /* map structure to the assemble language connection routine */
  /*         1) save current Data/Extra Segments               */
  /*         2) save old interrupt pointer                     */
  /*         3) assign new interrupt routine pointer           */
  /* --------------------------------------------------------- */
  SI          =  (struct SI_type *) SI_OVR;          /* ext. struct ptr.   */
  segread     (&segregs);                            /* get segment regs.  */
  SI[0].DSeg = segregs.ds;                           /* save current DS    */
  SI[0].old_ptr          = intr_table[intr_addr];    /* save origional int.*/
  SI[0].valid_no_of_func = max_func_num + 1;         /* include pc_time    */
  SI[0].func_table[0]    = (int *) pc_time;
  for (i=0; i<max_func_num; i++) {
    SI[0].func_table[i+1] = Function_Table[i];
  }

  /* initialize interrupt program */
  /* ---------------------------- */
  init_pc_time ();

  /* change the interrupt vector pointer to point to the asm.   */
```

```
  /* connection routine                                                */
  /* ----------------------------------------------------------------- */
  intr_table[intr_addr] =  (int *) SI_DVR;

  if (argc > 1) {
    if ( strcmpi ( argv[1], "-dos" ) == 0 )
      system ( "\command.com" );

    else if ( strcmpi ( argv[1], "-test" ) == 0 )
      test_func ();

  . else {
      strcpy ( argv[0], "" );
      forkv  ( "CADA-LDA", argv );
    }
  }
  else {
    system ( "CADA-LDA" );
  }

  intr_table[intr_addr] = SI[0].old_ptr;

  printf ("\x1b[2J System restored back to it's initial state\n");

} /* main() */
```



```
title   Software Interrupt Driver
page    ,132
; ------------------------------------------------------------- ;
;  Written by  : Alex Tsui                                      ;
;  Date        : 16:23:47  1/20/1987                            ;
;  Last Update : 20:38:14  1/27/1987                            ;
; ------------------------------------------------------------- ;
;  Title       : Call C function by Software Interrupt          ;
; ------------------------------------------------------------- ;

_TEXT   segment byte public 'CODE'
_TEXT   ends

_DATA   segment word public 'DATA'
_DATA   ends

CONST   segment word public 'CONST'
CONST   ends

_BSS    segment word public 'BSS'
_BSS    ends

; ------------------------------------------------------------- ;

DGROUP  group   CONST, _BSS, _DATA
        assume  CS: _TEXT, DS: _TEXT, SS: DGROUP, ES: DGROUP

; ------------------------------------------------------------- ;


func_num      equ     bx              ; set by user
no_of_params  equ     cx              ; set by user
err_flag      equ     ax              ; return with error code
PAGE          ;
```

```
_TEXT           segment

                public          _SI_DVR, SI_DVR
_SI_DVR         proc    far
SI_DVR:
                cli                             ; clear interrupt enable flag
                jmp     save_reg                ; jmp to start of routine

; ------------------------------------------------------------ ;
; Data Area                                                    ;
; ------------------------------------------------------------ ;
DSeg            dw      ?
Old_ptr         dd      ?
busy            dw      0000H
valid_func_no   dw      0000H
id_length       db      12
id              db      'ISAAC DRIVER '
params          dw      20 DUP (0000H)
func_table      dd      50 DUP (0000H)
                NOP
                NOP

; ------------------------------------------------------------ ;
; Code Area                                                    ;
; ------------------------------------------------------------ ;
save_reg:
                pushf
                push    si
                push    di
                push    ds
                push    es
                push    bp
                mov     bp,sp
                ;
PAGE            ;
```



```
int_rou:
                push    cs              ; establish local data segment ptr
                pop     ds
                ;
                mov     err_flag,00h    ; clear error flag
                ;
busy_check:
                mov     dx,busy
                cmp     dx,0001H        ; check for busy flag
                jne     not_busy        ; if busy then restore regs. & quit
                mov     err_flag,0011H  ; error code = 11H >> busy
                jmp     restore_regs
not_busy:       mov     busy,0001H      ; enable busy flag
                sti                     ; set interrupt enable flag
func_check:
                mov     dx,valid_func_no ; check for valid function selection
                cmp     func_num,dx
                jl      param_check
                ;
                mov     err_flag,0021H  ; error code = 21H >> invalid func. num.
                jmp     reset_busy
                ;
param_check:
                cmp     no_of_params,00H ; if no parametes then jump the next section
                je      calc_func_ofs
                cmp     no_of_params,20  ; check for valid # of parameters
                jle     push_param
                ;
                mov     err_flag,0031H  ; error code = 31H >> invalid no. of params
                jmp     reset_busy
                ;
push_param:
                mov     si,no_of_params ; push all parameter to stack (cx = no_of_params)
                dec     si              ;   last parameter first
                shl     si,01H
push_more:      mov     dx,params[si]
                push    dx
                dec     si
                dec     si
                loop    push_more
                ;
PAGE            ;
```

```
calc_func_ofs:
                mov     si,func_num     ; execute function wrt the func. table
                mov     cl,02H          ; multiply index by 4 to get func ptr.
                shl     si,cl
                ;
set_c_ds:
                mov     dx,DSeg         ; establish original C's Data seg. addr
                push    dx
                pop     ds
                ;
execute_function:                       ; call the C function
                call    dword ptr cs:func_table[si]
                ;
                cli
                ;
reset_busy:
                cli                     ; clear interrupt enable flag
                mov     cs:busy,00H     ; clear busy flag
                ;
restore_regs:
                mov     sp,bp
                pop     bp
                pop     es
                pop     ds
                pop     di
                pop     si
                popf
                iret
_SI_DVR         endp

_TEXT           ends

end
```



```
page            55,132
title           DOS4AH (pp_to_release: integer): integer
code            segment
                assume          cs:code

;       function called from Turbo Pascal, which releases PP_TO_RELEASE
;       paragraphs of memory from the current memory block using the DOS
;       function 4AH.  If this is possible, the the Turbo stack is
;       relocated to the high end of the new allocation and function
;       result is zero.  otherwise the DOS error code is returned.

;       Pascal declaration:
;       FUNCTION DOS4AH(PP_TO_RELEASE: INTEGER): INTEGER;

arguments       struc                   ; template to arguments.
save_bp         dw      ?               ; saved BP register.
ret_adr         dw      ?               ; return address (near call).
;.....................................................................
pp_to_release   dw      ?               ; integer # of paragraphs to release.
;.....................................................................
arguments       ends

dos4AH          proc    near
                push    bp
                mov     bp,sp           ; establish access to arguments
                push    ds              ; save Turbo's data segment.

                mov     ax,cs           ; calculate the paragraph size
                mov     bx,ss           ; of the current block as
                add     bx,1000H        ; SS+1000H minus code segment
                sub     bx,ax           ; address. Then calculate new size
                                        ; for memory block.
                sub     bx,[bp].pp_to_release
                mov     es,ax           ; segment address of current
                mov     ah,4AH          ; memory block.
                int     21H             ; ask DOS to deallocate the space.
                mov     ah,0
                jc      exit            ; was it possible to release the memory?

                mov     ax,ss           ; Yes:  set up the relocate stack
                mov     ds,ax           ; DS:SI is pointer to old stack
                sub     ax,[bp].pp_to_release
                mov     es,ax           ; ES:DI is the pointer to the new stack.
                mov     bx,sp
                mov     di,bx
                mov     si,bx

                mov     cx,sp           ; two's complement of SP is the stack
                neg     cx              ; size.
                cld                     ; move cx bytes from
                rep     movsb           ; DS:SI to ES:DI ...
                                        ; starting with the lowest byte
                mov     ss,ax           ; change the SS register ...
                xor     ax,ax           ; and set the error code to 0.

exit:           pop     ds
                pop     bp
                ret     4               ; Pop argument and "function result"

dos4AH          endp

code            ends

                end
```

```
page        55,132
title       DOS4BH (var program_name, parameter_string): integer
code        segment
            assume      cs:code

; FUNCTION: called from Turbo Pascal, executes .COM
; or .EXE programs using the DOS function 4BH.
; Preserves the stack registers, returning the
; error code 0 if successfull or the DOS error code
; if the execute fails.

; Pascal declaration:
; FUNCTION DOS4BH (var program_name, parameter_string): integer

arguments   struc                       ; template to arguments.
save_bp     dw          ?               ; saved BP register.
ret_adr     dw          ?               ; return address (near call).
;-------------------------------------------------------------------------
param_str   dd          ?               ; DWORD pointer to param. string
prog_name   dd          ?               ; DWORD pointer to ASCIIZ
                                        ; program name.
;-------------------------------------------------------------------------/----
arguments   ends

dos4BH      proc        near
            jmp         short exec_code
;---------------------------------Load control block----------------------
lcb_envir   dw          0               ; segment address of environment
lcp_ps_o    dw          ?               ; offset of parameter string.
lcb_ps_s    dw          ?               ; segment of parameter string.
lcb_fcb1    dd          0               ; DWORD pointer to FCB1.
                                        ; (5CH in PSP).
lcb_fcb2    dd          0               ; DWORD pointer to FCB2
                                        ; (6CH in PSP)
;---------------------------------Local variables-------------------------
ss_save     dw          ?               ; saved stack segment.
sp_save     dw          ?               ; saved stack pointer.
;-------------------------------- ----------------------------------------
exec_code:  push        bp
            mov         bp,sp           ; establish access to arguments.
            push        ds              ; save the Turbo data stack.
            call        relo1           ; set BX to relocation
                                        ; factor: true offset ...
relo1:      pop         bx              ; of relo1 minus local offset
            sub         bx,offset relo1

            les         di,[bp].param_str ; get offset:segment
                                        ; of parameter string.
            mov         ax,es
            mov         cs:lcb_ps_s[bx],ax ; save parameter
                                        ; string segment ...
            mov         cs:lcp_ps_o[bx],di ; ... and ofset in LCB.
            mov         ax,ss           ; save SS and SP registers
            mov         cs:ss_save[bx],ax ;
            mov         ax,sp           ;
            mov         cs:sp_save[bx],ax ;
            push        cs              ; set up ES:BX as pointer
                                        ; to the LCB ...
            pop         es              ;
            add         bx,offset lcb_envir
            lds         dx,[bp].prog_name ; and DS:DX as pointer
                                        ; to the ASCIIZ ...
            inc         dx              ; ... program name.
            mov         ax,4B00H        ; DOS EXEC FUNC: al=0, ah=4BH
            int         21H             ; let her rip.

            sti                         ; disable interrupts while
                                        ; stack messed up
            jc          run_error       ; sucessful execution ?
            xor         al,al           ; Yes, set error code = 0.
```



```
run_error:  xor         ah,ah           ;
            call        relo2           ; recalculate relocation factor ...
relo2:      pop         bx              ;
            sub         bx,offset relo2 ;
            mov         dx,cs:ss_save[bx] ; ... and restore the stack.
            mov         ss,dx           ;
            mov         dx,cs:sp_save[bx] ;
            mov         sp,dx           ;
            cli                         ; interrupts back on ... all done!

            pop         ds
            pop         bp
            ret         10              ; pop arguments and "function result"
dos4BH      endp
code        ends
            end
```

```
{=============================================================}
{ Oct 11, 86                  "get current dir & path name"    }
{=============================================================}

{ ---------------------------------------------------------- }
procedure release_heap (pp_to_release: integer);
  label finish;
  var   memory_segment : integer;

  begin
    if dos48H (pp_to_release, memory_segment) = 0 then begin
      if dos_error_check (dos49H (memory_segment)) then goto finish;
    end
    { --------------- if not, then reduce the size of the current allocation }
    else begin
      if dos_error_check (dos4AH (pp_to_release)) then goto finish;
    end;

    finish:
  end;


procedure init_dos (block_to_reserve : integer);
  var mem : integer;
  begin
    { ---------- reserve internal storage and release the rest from the HEAP }
    release_heap (block_to_reserve);
  end;


type dos_string = string[128];

procedure cli_dos (var parameter_string : dos_string);
  label finish;
  var   program_name,  prompt_str       : ascii1;
        mem_avail_in_program, dum_int   : integer;
        opt_loc, par_loc                : integer;
        pause                           : boolean;
        dum_key                         : char;

  begin
    { ---------------------------------- terminate if can't get comspec name }
    if get_comspec (program_name) then begin
      writeln (bell, 'Missing COMSPEC');
      goto finish;
    end;

    { ------------------------------------------- analysis command string }
    pause := true;

    opt_loc := pos ('-p', parameter_string);
    dum_int := pos ('-P', parameter_string);
    if ( (dum_int > 0) and (dum_int < opt_loc) ) then
      opt_loc := dum_int;

    par_loc := pos (':=', parameter_string);
    if (opt_loc = 0) then
      pause := true
    else if ( (par_loc = 0) and (opt_loc > 0) ) then
      pause := false
    else if ( (par_loc > 0) and (opt_loc < par_loc) ) then
      pause := false;

    if par_loc = 0 then
      parameter_string := program_name
    else begin
      delete (parameter_string, 1, par_loc+1);
      parameter_string := parameter_string;
    end;
```



```
    parameter_string := '/c ' + parameter_string;

    { ----------- *
    writeln ('DOS > ', parameter_string);
    *----------- }

    parameter_string [length (parameter_string)+1] := #13;

    if dos_error_check (dos49H (program_name, parameter_string)) then
      writeln ('Command Cancelled.');

    finish:
      if (pause) then begin
        gotoXY (49,25);
        write  (' Press RETURN to continue .....');
        read   (dum_key);
      end;
  end;
```

```
{ - - - - DO LOOP INTERPRETER ROUTINES - - - - - - - - - - - - - - - - - }
TYPE  do_ptr      = ^do_record;
      do_record   = RECORD
                      text    : st128;
                      next    : do_ptr;
                    END;
      do_stru     = RECORD
                      lab     : integer;
                      ctr     : integer;
                      first   : do_ptr;
                    END;

VAR   _do_rec         : do_ptr;
      _do             : array [0..9] of do_stru;
      _do_flg,
      _do_stored_flg : boolean;
      _do_level       : integer;
      _do_first,
      _do_last,
      _do_rec_ptr     : do_ptr;
      _do_cur_lab     : integer;
{------------------------------}

Procedure Dispose_All_Loop;
  VAR   temp_ptr : do_ptr;
  BEGIN
     _do_flg := FALSE;
    While (_do[0].first <> nil) do begin
       temp_ptr := _do[0].first;
       _do[0].first := temp_ptr^.next;
       DISPOSE (temp_ptr);
    end;
  END;

{.pa}

Function cli_STORE_DO (Var buf : st128) : boolean;
  VAR  data            : st10;
       buf128          : st128;
       v_code, I_data  : integer;
       token3          : st3;
       label_begin,
       label_end       : integer;
       temp_ptr        : do_ptr;
       qrror           : boolean;

  BEGIN
   Push_msg (' Proc: cli_Store_DO');
     _do_flg        := TRUE;
     _do_level      := -1;
     _do_cur_lab    := -1;
     _do_stored_flg := FALSE;

     NEW (_do_rec);
     _do_rec^.text := buf;
     _do_rec^.next := nil;
     _do_first     := _do_rec;
     _do_last      := _do_rec;

     error := false;

     buf128 := copy(buf, 4, length(buf)-3);
     IF (FoundData (buf128, data)) then begin
        VAL (data, I_data, v_code);
        IF (v_code = 0) then
           label_begin := I_data
        ELSE begin
           AbortCLI ('invalid LABEL entry');
           error := true;
        end;
```



```
        end
        ELSE begin
           AbortCLI ('no LABEL are specified');
           error := true;
        end;

        if (not error) then begin
          REPEAT
            Get_Input (' DO', buf);
            NEW (_do_rec);
            _do_rec^.text  := buf;
            _do_rec^.next  := nil;
            _do_last^.next := _do_rec;
            _do_last       := _do_rec;

            token3 := buf;
            IF (token3 = 'END') then begin
              buf128 := COPY(buf, 5, LENGTH(buf)-4);
              IF (FoundData (buf128, data)) then begin
                VAL (data, I_data, v_code);
                IF (v_code = 0) then
                  label_end := I_data
                ELSE begin
                  AbortCLI ('invalid LABEL entry');
                  error := true;
                end;
              end
              ELSE begin
                AbortCLI ('no LABEL are specifiqd');
                error := true;
              end;
              IF (label_begin = label_end) then
                _do_stored_flg := TRUE;
            end;
          UNTIL ( _do_stored_flg or error);

          printfln ('');

          _do_rec_ptr := _do_first;
        end;

        if error then begin
          Dispose_all_loop;
          cli_store_do := false;
        end
        else
          cli_store_do := true;
      Pop_msg;
    END; { cli_STORE_DO }

{.pa}

Procedure cli_Begin_DO (buf : st128);
  VAR  com, data     : st10;
       I_data, v_code : integer;
       err            : byte;
  BEGIN
   Push_msg (' Proc: cli_Begin_DO');
     err := 0;
     IF (FoundData (buf, data)) then begin
        VAL (data, I_data, v_code);
        IF (v_code = 0) then begin

           IF (_do_cur_lab <> I_data) then begin
              _do_level := _do_level + 1;
              _do[_do_level].lab := I_data;
              _do_cur_lab := I_data;
              _do[_do_level].first := _do_rec_ptr;

              IF (FoundCom (buf, '=', com)) then begin
```

```
                IF (com[1] = 'I') then begin

                   IF (FoundData (buf, data)) then begin
                      VAL (data, I_data, v_code);
                      IF (v_code = 0) then
                         _do[_do_level].ctr := I_data
                      ELSE err := 1;
                   end
                   ELSE err := 2;
                end
                ELSE err := 3;
             end
             ELSE err := 2;
          end;
       end
       ELSE AbortCLI ('invalid DO LOOP LABEL');
    end
    ELSE AbortCLI ('missing LABEL');
    DisplayError (err);
  Pop_msg;
END; { cli_BEGIN_DO }

{.pa}

Procedure cli_End_DO (buf : st128);
  VAR  com, data      : st10;
       I_data, v_code : integer;
       temp_ptr       : do_ptr;
  BEGIN
   Push_msg (' Proc: cli_End_DO');

     IF (FoundData (buf, data)) then begin
        VAL (data, I_data, v_code);
        IF (v_code = 0) then begin

           IF (_do_cur_lab = I_data) then begin
              _do[_do_level].ctr := _do[_do_level].ctr - 1;

              IF (_do[_do_level].ctr <= 0) then begin
                 _do_level := _do_level - 1;

                 IF (_do_level >= 0) then begin
                    _do_cur_lab := _do[_do_level].lab;
                 end
                 ELSE
                    Dispose_All_Loop;
              end
              ELSE _do_rec_ptr := _do[_do_level].first;
           end
           ELSE AbortCLI ('DO: label mismatch');
        end
        ELSE AbortCLI ('invalid DO LOOP LABEL');
     end
     ELSE AbortCLI ('missing LABEL for the END statement');
   Pop_msg
  END; { cli_END_DO }

{.pa}

Procedure cli_WAIT (token   : st3;                { sub command token }
                    Var buf : st128);             { command line      }
  Var   data, time              : st10;
        current_time, wait_time : string[8];
        time2                   : st2;
        r_data                  : real;
        v_code, err, i          : integer;
        key                     : char;
        r_hr,r_min,r_sec        : integer;
        v_code1,v_code2,v_code3 : integer;
  BEGIN
```



```
   Push_Msg (' Proc: cli_wait');
     err := 0;
     IF (token = 'KEY') or (token = 'TIM') or (token = 'DEL' ) then begin

        IF (token = 'KEY') then begin
          i := whereY;
          gotoXY(1,1);       clrEOL;
          textColor(blink); printf (' Press any key to continue ');  NormVideo;
          read (kbd, key);
          Open_user_window;
          gotoXY(1,1);       clrEOL;
        end
        ELSE IF ( FoundData(buf, data) ) then begin
           IF (token = 'TIM') then begin
              r_hr := -1;   r_min := -1;   r_sec := -1;
              IF (length(data) = 8) then begin
                 wait_time :=   COPY(data, 1, 2) + ':'
                              + COPY(data, 4, 2) + ':'
                              + COPY(data, 7, 2);
                 FOR i:=1 to 8 do
                    IF (wait_time[i] = ' ') then   wait_time := '0';
              end;
              IF ( (wait_time < '00:00:00') or (wait_time > '23:59:59') ) then
                 AbortCLI ('invalid time spec.: HH:MM:SS');

              REPEAT
                 current_time := Get_Time;
                 update_time_date (false);
              UNTIL (keypressed) or (current_time >= wait_time);
              printfln ('');
           end
           ELSE IF (token = 'DEL') then begin
              VAL (data, r_data, v_code);
              IF (v_code = 0) then
                 DELAY ( TRUNC( r_data * 1000 ) )
              ELSE  err := 1;
           end
           ELSE err := 1;
        end
        ELSE  err := 2;
     end
     ELSE  err := 3;
     DisplayError (err);
   Pop_msg;
  END; { cli_WAIT }

{.pa}

Procedure cli_EXECUTE_1 (token   : st3;           { sub command token }
                         Var buf : st128);        { command line      }
  Var   data           : st10;
        r_data         : real;
        v_code, err    : integer;

  BEGIN
   Push_msg (' Proc: cli_EXECUTE_1');
     err := 0;
     IF (token = 'FIL') or (token = 'USE') or (token = 'PAR') then begin
        IF ( FoundData(buf, data) ) then begin
           CASE (token[1]) of
             'F' : begin
                      ASSIGN (cmd_file[cmd_num +1], data);
                      {$I-}  RESET (cmd_file[cmd_num +1]);  {$I+}
                      IF (IOresult <> 0) then
                         printfln (' CANNOT OPEN CMD_FILE: ' + data)
                      ELSE begin
                         cmd_num := cmd_num + 1;
                         cmd_file_exist[cmd_num] := TRUE;
                      end;
                   end;
```

```
            ELSE  err := 1;
          end;
        end
        ELSE  err := 2;
      end
      ELSE  err := 3;
      DisplayError (err);
    Pop_msg;
  END; { cli_EXECUTE_1 }
```



```
Procedure cli_HELP ;
   VAR help_file : text;
       ch        : char;
       buf       : st128;
   BEGIN
     if (_help_loadnd) then
       Change_Screen (help, $croll)
     else begin
       ASSIGN (help_file, 'CADA-LDA.HLP');
       {$I-}  RESET(help_file);  {$I+}
       IF (IOresult <> 0) then begin
          printfln ('HELP file does not exist in current directory');
          printfln ('');
       end
       ELSE begin
         if (NOT _help_opened) then begin
           _help_opened := true;
           Init_Window (W[help], true, false);
         end;

         _help_loaded := true;
         With W[help] do begin
           TextColor      (foreground);
           TextBackGround (background);
           Ram_Page       (handle);
           WHILE (not EOF(help_file)) DO begin
             READLN (help_file, buf);
             printfln (buf);
           end;
           CLOSE (help_file);
           Appear_Window  (W[help]);
           Change_Screen  (help, $croll);
         end;
       end;
     end;
     Open_User_Window;
   END;
```

```
{ ------------------------------------------- }
{ cli routine for the LDA unit                }
{ ------------------------------------------- }

procedure INIT_CLI_LDA;
  var i : byte;
  begin
    with _LDA do begin
      mode     := 0;        DispMode := True;
      TimeBase := 1e-3;     S        := 2;
      nsets    := 0;        size     := 0;        Isize    := 0;
      SetRead  := 0;
      Interval := 0;        DelayFactor := 1.0;
      dma      := 3;
      data     := nil;
    end;
    enable.LDA := false;
    INIT_LDA_INTERFACE (6, 'P');
  end; { procedure INIT_CLI_LDA }


procedure Gen_LDA_Data (mode, i : integer; Var P1, P2, P3, P4 : real);
  begin
    with _LDA.data^[i] do begin
      P1 := _LDA.TimeBase * Tm * abs(1.0 * ($0001 shl (TDx shr 4)));
      P3 := 0;
      P4 := 0;
      case mode of
        0 : begin
              P1 := Pm * abs(1.0 * ($0001 shl (Px shr 4)));
              P2 := F;
              P3 := Tm * abs(1.0 * ($0001 shl (TDx shr 4)));
              P4 := Dm * abs(1.0 * ($0001 shl (TDx and $0F)));
            end;
        1 : begin
              P2 := 14.95959712 * Dm * abs(1.0 * ($0001 shl (TDx and $0F)));
            end;
        2 : begin
              P2 := 14.95959712 * Dm * abs(1.0 * ($0001 shl (TDx and $0F)));
              P3 := F;
            end;
        3 : begin
              P2 := 1e-9 * Pm * abs(1.0 * ($0001 shl (Px  shr 4)));
              P3 := F;
            end;
        4 : begin
              P2 := 1e-9 * Pm * abs(1.0 * ($0001 shl (Px  shr 4)));
            end;
      end;
    end;
  end; { procedure Gen_LDA_Data (mode, i, P1, P2, P3, P4) }

{.pa}

function DispData ( value : real ) : st80;
  var v : real;
  begin
    v := abs (value);
    if      v <= 1e3 then  DispData := R_str (value, 14, 9)
    else if v <= 1e6 then  DispData := R_str (value, 14, 6)
    else                   DispData := R_str (value, 14, 3);
  end; { function DispData (value) : st80 }


function LDA_title_str (mode           : byte;
                        fill           : st80) : st_type;
  var line : st_type;
  begin
    line := fill + 'Mode_' + B_str(_LDA.Mode, 1) + fill;
    case _LDA.Mode of
```



```
      0 : line := line + S_str('P','C',14,'_') + fill
                       + S_str('F','C',14,'_') + fill
                       + S_str('T','C',14,'_') + fill
                       + S_str('D','C',14,'_') ;
      1 : line := line + 'Interval_(sec)' + fill + '_Doppler_(cps)' ;
      2 : line := line + 'Interval_(sec)' + fill + '_Doppler_(cps)' + fill
                       + '__Fringe_Count' ;
      3 : line := line + 'Interval_(sec)' + fill + 'Burst_t._(sec)' + fill
                       + '__Fringe_Count' ;
      4 : line := line + 'Interval_(sec)' + fill + 'Transit_t(sec)' ;
    end;
    LDA_title_str := line;
  end; { function LDA_title_str (mode, fill) : st_type }


function LDA_param_str (mode           : byte;
                        index          : integer;
                        P1, P2, P3, P4 : real;
                        fill           : st80) : st_type;
  var line : st_type;
  begin
    line := fill + I_str(index+1, 7) + fill
                 + DispData(P1)      + fill + DispData(P2);
    case _LDA.mode of
      0 : line := line + fill + DispData(P3) + fill + DispData(P4);
      2 : line := line + fill + DispData(P3);
      3 : line := line + fill + DispData(P3);
    end;
    LDA_param_str := line;
  end; { function LDA_param_str (mode, index, P1, P2, P3, P4, fill) : st_type }


function LDA_Output (title : boolean; index : integer ) : st_type;
  Var   P1, P2, P3, P4 : real;
        line           : st_type;
  begin
    if (title) then
      LDA_Output := LDA_title_str (_LDA.mode, ' ')
    else begin
      Gen_LDA_Data (_LDA.mode, index, P1, P2, P3, P4);
      LDA_Output := LDA_param_str (_LDA.mode, index, P1, P2, P3, P4, ' ');
    end;
  end; { function LDA_Output (title : boolean) : st_type }

{.pa}

procedure Disp_LDA_Results;
  Var   i,  ii          : integer;
        P1, P2, P3, P4 : real;
        ch              : char;
  begin
    if (Not _LDA.DispMode) then exit;

    Open_LDA_Window;
    ClrScr;
    GotoXY (1,1);
    printf ( S_str('', 'R', W[LDA].Xview+1, #240) );

    GotoXY (1,1);
    printf ( LDA_title_str (_LDA.mode, #240+#240) );

    with W[LDA] do Window (Xpos, Ypos+1, Xpos + Xview, Ypos + Yview);
    GotoXY (1,1);
    i := 0;
    while (i < _LDA.SetRead) do begin
      if keypressed then begin
        read (kbd, ch);
        ii := _LDA.SetRead - W[LDA].Yview;
        if (ii > i).then i := ii;
      end;
```

```
    if (i > 0) then  printfln ('');

    Gen_LDA_Data (_LDA.mode, i, P1, P2, P3, P4);
    printf ( LDA_param_str (_LDA.mode, i, P1, P2, P3, P4, '  ') );
    i := i + 1;
  end;
  with W[LDA] do Window (Xpos, Ypos, Xpos + Xview, Ypos + Yview);
  Open_USER_Window;
end; { procedure Disp_LDA_Results }

{.pa}

Procedure cli_LDA   (token   : st3;                { sub command token  }
                     Var buf : st128);             { command line       }
  Var   data    : st10;
        data2   : string[2];
        r_data  : real;
        v_code,
        err     : integer;
        Isize   : integer;

BEGIN
 if (NOT enable.LDA) then  exit;

 Push_Msg ('Proc: cli_LDA');
   err   := 0;
   data2 := buf;
   IF   (token='NSE') or (token='INT') or (token='DMA')
     or (token='MOD') or (token='TIM') or (token='DEL') then begin
      IF ( FoundData( buf, data ) ) then begin
         VAL (data, r_data, v_code);
         IF (v_code = 0) then begin
           with _LDA do begin

             IF      (token = 'NSE') then begin
               if (r_data >= 0) and (r_data <= max_lda_array) then
                    nsets := TRUNC(r_data)
               else nsets := max_lda_array;

               _LDA.size := _LDA.nsets * 6.0;

               if (_LDA.Isize <> 0) then
                 FreeMem (_LDA.data, _LDA.Isize);

               if (_LDA.size >= 0) and (_LDA.size <= 32767.0) then
                 _LDA.Isize := trunc(_LDA.size)
               else if (_LDA.size > 32767.0) and (_LDA.size <= 65535.0) then
                 _LDA.Isize := TRUNC (_LDA.size - 65535.0);

               GetMem (_LDA.data, _LDA.Isize);
               _LDA.SetRead := 0;
             end

             ELSE IF (token = 'INT') then begin
               if (r_data <= 32.0) then  Interval := r_data
               else                      Interval := 32.0;
             end

             ELSE IF (token = 'DMA') then begin
               if ( TRUNC(r_data) = 1 ) then  dma := 1
               else                           dma := 3;
             end

             ELSE IF (token='MOD') then begin
               if (r_data >= 0) and (r_data <= 4) then
                   _LDA.mode := Lo ( Trunc (r_data) )
               else _LDA.mode := 0;
             end
```



```
             ELSE IF (token='TIM') then begin
               _LDA.S := Trunc (r_data);
               case _LDA.S of
                 0  : _LDA.TimeBase := 1;
                 2  : _LDA.TimeBase := 1e-3;
                 3  : _LDA.TimeBase := 100e-9;
                 else begin
                      _LDA.TimeBase := 0;
                      _LDA.S        := 1;
                 end;
               end;
             end

             ELSE IF (token = 'DEL') then begin
               DelayFactor := r_data;
             end;
           end;
         end

         ELSE  err := 1;
       end

      ELSE IF (token='DIS') then begin
        if      (data2 = 'ON') then  _LDA.DispMode := True
        else if (data2 = 'OF') then  _LDA.DispMode := False
        else if (data2 = 'AS') then  Disp_LDA_Results;
      end

      ELSE  err := 3;
      DisplayError (err);
   Pop_Msg;
 END; { cli_LDA }

{.pa}

procedure Read_LDA;
  begin
    if (_LDA.data <> nil) and
       (_LDA.nsets <> 0)
    then begin
      if (_LDA.Interval = 0) then begin
        INIT_LDA_INTERFACE (6, 'D');
        INIT_and_RUN_DMA;
        WAIT_TILL_COMPLETE ('D');
      end
      else begin
        INIT_LDA_INTERFACE (6, 'P');
        WAIT_TILL_COMPLETE ('P');
      end;
      Disp_LDA_Results;
    end;
```

```
Var _trav : array[0..3] of record
              posn  : real;
              tol   : real;
              wait  : byte;
              delay : byte;
              disp  : boolean;
            end;
    _trav_dev      : byte;
    _trav_offs     : real;
    _tra_cha       : byte;
    _tra_err       : boolean;


Procedure init_cli_trav;
  var i : byte;

  begin
    for i:=0 to 3 do
      with _trav[i] do begin
        posn := 0.0;  tol := 0.1;  wait := 100;  delay := 50;
      end;
    _trav_offs := 0.01667;                    { error correction constant }
    _trav_dev  := 0;
    enable.trav := false;
  end;


Procedure cli_TRAV (token   : st3;              { sub command token }
                    Var buf : st128);           { command line      }
  Var   data   : st10;
        data3  : st3;
        r_data : real;
        v_code,
        err, i : integer;
        inc_flg: boolean;

  Procedure Wait_Trav (com : st10);
    VAR   ctr, i              : integer;
          new_posn, posn, tol : real;
          old_posn            : array [0..2] of real;
          finish              : boolean;
    BEGIN
       if (NOT enable.trav) then exit;

       Push_Msg (' Proc: Wait_Trav');
          finish := FALSE;
          _tra_err := FALSE;
          ctr := 0;
          new_posn := _trav[_tra_cha].posn;
          tol      := _trav[_tra_cha].tol;
          FOR i:=0 to 3 DO
             old_posn[i] := 0;
          REPEAT
             IF (upcase(com[1]) = 'A' ) then begin
                finish := TRUE;
                FOR i:=0 to 2 DO begin
                   TRAVER (read_d, i, new_posn);
                   _trav[i].posn := new_posn;
                   IF (new_posn <> old_posn[i]) then
                      finish := FALSE;
                   old_posn[i] := new_posn;
                end;
                UpdateWindow;
             end
             ELSE begin
                TRAVER (read_d, _tra_cha, posn);
                _trav[_tra_cha].posn := posn;
                IF (debug_488 > 5) then
                   writeln ('posn read d/w = ', posn:9:2);
                IF ((posn < new_posn + tol) and (posn > new_posn - tol)) then
                   finish := TRUE
                ELSE begin
                   UpdateWindow;
                   ctr := ctr + 1;
                end;
                IF ( (upcase(com[1]) = 'D') and (ctr > _trav[_tra_cha].delay) )
                   or
                   ( (upcase(com[1]) = 'W') and (ctr > _trav[_tra_cha].wait) )
                   then begin
                      write (bell);
                      IF (debug_488 > 5) then
                         writeln ('traver error in wait/delay');
                      _tra_err := TRUE;
                end;
             end;
          UNTIL (finish or _tra_err);
          UpdateWindow;

       Pop_Msg;
    END;
```



```
{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -}
BEGIN
 Push_Msg (' Proc: cli_TRAV');
   err := 0;

   IF ( token = 'X') or (token = 'Y') or (token = 'Z') or (token = 'R')
      or (token = '') or (token = 'DEV') or (token = 'DEL') then begin

      IF ( FoundData(buf, data) ) then begin
         data3 := data;

         IF (token[1] in ['X', 'Y', 'Z', 'R']) then begin
            if (token[1] = 'R') then
               _tra_cha := 3
            else
              _tra_cha := ABS ( INTEGER('X') - INTEGER(token[1]) );

            IF ( UPCASE (data[1]) = 'I') then begin
               inc_flg := TRUE;
               DELETE (data, 1, 1);
            end
            ELSE
               inc_flg := FALSE;

            VAL (data, r_data, v_code);

            IF (v_code = 0) then begin

               IF (inc_flg) then
                  _trav[_tra_cha].posn := _trav[_tra_cha].posn
                                          + r_data
               ELSE
                  _trav[_tra_cha].posn :=   r_data;

               r_data := _trav[_tra_cha].posn;

               IF (r_data > 0) then
                  r_data := r_data + _trav_offs
               ELSE
                  r_data := r_data - _trav_offs;

               DELAY (300);
               TRAVER (set_d, _tra_cha, r_data);
            end
            ELSE  err := 1;
         end

         ELSE IF (token = 'DEL') then begin
```

```
            IF (data[1] = 'W') then
               Wait_Trav ('wait')
            ELSE IF (data[1] = 'A') then
               Wait_Trav ('all finish')
            ELSE begin
               VAL (data, r_data, v_code);

               IF (v_code = 0) then begin
                  DELAY ( TRUNC(r_data * 1000.0));
                  Wait_Trav ('delay');
               end
               ELSE
                  err := 1;
            end;
         end

         ELSE IF (token = '') and (data3 = 'INI') then begin
            FOR i:=0 to 2 DO begin
               _trav[i].posn := 0.0;

               TRAVER (clear_d, i, _trav[i].posn);
            end;
            UpdateWindow;
         end

         ELSE IF (token = 'DEV') then begin
           Val (data, r_data, v_code);
           if (v_code = 0) then
             _trav_dev := trunc (r_data)
           else
             err := 1;
         end;
      end
      ELSE err := 2;
    end
    ELSE err := 3;
    DisplayError (err);
  Pop_Msg;
 END; { cli_TRAV }
{.pa}
```



```
{-------------------------------------------------------------------}
type st10    = string[10];
     array16 = array[0..31] of byte;
VAR  debug_488   : byte;          {-depended on it's value, various    }
                                  { level of display will be generated }

{-------------------------------------------------------------------}
{ Initialize IEEE 488 I/O Routine                                     }
{ ----------------------------                                      }
{  1) INIT_GPIB (gpib_name   : st40;     { I   : GPIB name in ibconf  }
{                var gpib_id : integer   { O   : GPIB device ID #     }
{                var error   : st10)     { O   : error message        }
{               : boolean;               { ture if error occure       }
{-------------------------------------------------------------------}

procedure ib_error (ib_name : st10; err_msg : st128);
   begi
     (* This routine would, among other things, check iberr to
        determine the exart cause of the error condition and then
        take action appropriate to the application.
        For errors during data transfers, ibcnt may be examined
        to determine the actual number of bytes transferred.
      *)
     err_msg := ib_name
                + ' ibsta=' + hex(ibsta)
                + ' iberr=' + hex(iberr)
                + ' ibcnt=' + hex(ibcnt);
   end;
type gpib_ntype = string[40];

function INIT_GPIB (gpib_name   : gpib_ntype; { I   : GPIB name in ibconf }
                    var gpib_id : integer;    { O   : GPIB device ID #    }
                    var error   : st128)      { O   : error message       }
                    : boolean;                { ture if error occure      }

BEGIN
  error := '';
  gpib_id := ibfind (gpib_name);
  if (gpib_id < 0) then begin
    init_gpib := true;
    error := 'ibfind (' + gpib_name + ')';
  end
  else begin
    ibclr (gpib_id);   ib_error ('ibclr', error);
    if (length(error) > 0) then
      init_gpib := true
    else
      init_gpib := false;
  end;
END; { Proc: INIT_GPIB }
{.pa}

{-------------------------------------------------------------------}
{ ==== read data from National Instruments GPIB_PCIIA interface,    }
{      at base I/O address $02E1                                    }
{                                                                   }
{      Convert the BUS_COMMAND, DEVICE_ADDRESS, & DATA information  }
{      to a format which one can transfer it to the IEEE-488 bus.   }
{      This info. is received by the CTA IEEE-488 controller and    }
{      thus converted to information the CTA can understand.        }
{                                                                   }
{  1) Write_pc (GPIB_id   : Integer;    I = GPIB address            }
{               Bus_Com   : Integer;    I = $30 -> Write & set Remote}
{               Dev_Addr  : Integer;    I = Dev address (12 bits)   }
{               Data      : Byte);      I = 8 bits data             }
{                                                                   }
{ ==== read data form the DISA IEEE 488 card                        }
{  2) READ_PC (GPIB_id,                       { I GPIB address      }
{              Var num_of_byte: byte;         { I # of byte to read }
{              Var data   : array16;          { O data read from 488 }
```

```
{              err_msg    : st128)            { O error message      }
{              : boolean;                     { true if error occur  }
{                                                                    }
{--------------------------------------------------------------------}

PROCEDURE WRITE_PC (GPIB_id, BUS_COM, DEV_ADDR :INTEGER;  DATA :BYTE );

VAR
   FORMAT_A, FORMAT_B :INTEGER;

BEGIN
  Push_msg (' Proc: Write pc');
    FORMAT_A := $C000
              + (BUS_COM AND $3F) SHL 8
              + (DEV_ADDR SHR 5);
    FORMAT_B := $8000
              OR (DEV_ADDR AND $0018) SHL 9
              OR (DEV_ADDR AND $0007) SHL 4
              OR (DATA AND $F0) SHL 4
              OR (DATA AND $0F);

    {**** SEND FORMAT B (lo, hi) then A (lo, hi) ****}
    ibbuf[1] := chr( lo( format_b));
    ibbuf[2] := chr( hi( format_b));
    ibbuf[3] := chr( lo( format_a));
    ibbuf[4] := chr( hi( format_a));

    ibwrt (gpib_id, ibbuf, 4);
  Pop_msg;
END; { Proc: WRITE_PC }
{.pa}

{ ==== read data form the DISA IEEE 488 card }
Function READ_PC (GPIB_id        : integer;        { I GPIB address       }
                  num_of_byte    : byte;           { I # of byte to read  }
                  Var data       : array16;        { O data read from 488 }
                  Var err_msg    : st128)          { O error message      }
                  : boolean;                       { true if error occur  }

  VAR max, addr, i, j             : integer;
      er_format, er_seq, er_valid : boolean;
      temp1, temp2                : byte;

  Function Validate_data : boolean;
    BEGIN
      max := (num_of_byte shr 1) - 1;          { max = (num_of_byte / 2)-1 }
      er_format := false;
      er_seq    := false;
      er_valid  := false;
      j := 1;

      For i:=0 to max do begin
        temp1 := integer (ibbuf[j]);
        temp2 := integer (ibbuf[j+1]);

        data[i] := (temp1 and $0F) or (temp2 shl 4);

        addr := ((temp2 shr 1) and $08) or ((temp1 shr 4) and $07);

        If ((addr and max) <> (max - i)) then     { check for addr error    }
          er_seq := true;

        If ((temp2 and $20) = 0) then             { check for data valid    }
           er_valid := false;  { <<-------------- this should be set to TRUE }

        If (i < max) then begin
          If ((temp2 and $C0) <> $80) then .      { check for FORMAT B word }
            er_format := true;
        end
        Else begin
```



```
          If ((temp2 and $C0) <> $C0) then        { check for FORMAT A word }
            er_format := true;
        end;
        j := j + 2;
      end;

      Validate_data := not (er_format or er_seq or er_valid);
    END; { Func: Validate_data : boolean }

  var rsp : integer;

  BEGIN
    Push_msg (' Func: READ_PC ');
      err_msg := '';
      ibrsp (gpib_id, rsp);
      if ((rsp and $FF) <> $22) then begin
        Read_PC := true;
        err_msg := 'ibrsp=' + Hex(rsp);
        ibtrg (gpib_id);
        ibclr (gpib_id);
      end

      else begin
        ibrd (gpib_id, ibbuf, num_of_byte);

        if (not validate_data) then begin
          Read_PC := true;
          If er_format then
            err_msg := 'Format ';
          if er_seq then
            err_msg := err_msg + 'Sequence ';
          if er_valid then
            err_msg := err_msg + 'Invalid Data';
        end

        else begin
          Read_PC := false;
        end;
      end;
    Pop_msg;
  END; { Read_PC (..) : boolean }
{.pa}
```

```
{ ------------------------------------------------------- }
{  Written by  : Alex Tsui                                }
{  Date        : 23:04:34  1/21/1987                      }
{  Last Update : 23:03:26  2/19/1987                      }
{ ------------------------------------------------------- }
{  ISAAC Driver Routines ( make use of INT 60 )           }
{ ------------------------------------------------------- }

const isaac_drv_copyright = 'Copyright 1987 TSUI''s Co.';

const intr_addr                 = $60;
      ISAAC_id_string : string[15] = 'ISAAC DRIVER';

type  SI_rec    = record
                    dummy1  : array[0..13] of byte;
                    ID      : string[13];
                    param   : array[0..19] of integer;
                  end;

var   SI_intr   : ^SI_rec absolute $0000:$0180;  { at INT 60 }


function RUN_ISAAC : integer;
var error : integer;
begin
  if (SI_intr^.ID = ISAAC_id_string ) then begin
    intr (intr_addr, regs);
    error := regs.ax;
  end
  else begin
    error := $FF;
    printfln (Bell + 'ISAAC DRIVER not found');
  end;
  if (error <> 0) then printfln (Bell + 'ISAAC DRIVER error: '+ I_str(error,7));
  RUN_ISAAC := error;
end;

{.pa}

{ ------------------------------------------------------- }
function SI_time : integer;
begin
  regs.bx := 0;                         { function num      }
  regs.cx := 0;                         { no of param words }
  SI_time := RUN_ISAAC;
end;

function AINSC ( low_channel, high_channel, count, rate : integer;
                 var target, opt ) : integer;
var  option : string[255] absolute opt;
begin
  option[length(option)+1] := #0;       { convert PASCAL string to C format }
  regs.bx := 4;                         { function num for AINSC() }
  regs.cx := 10;                        { no of param words        }
  SI_intr^.param[0] := low_channel;     { low  channel #  (16 bit) }
  SI_intr^.param[1] := high_channel;    { high channel #  (16 bit) }
  SI_intr^.param[2] := count;           { LSB = count     (32 bit) }
  SI_intr^.param[3] := 0;               { MSB = 0                  }
  SI_intr^.param[4] := rate;            { LSB = rate      (32 bit) }
  SI_intr^.param[5] := 0;               { MSB = 0                  }
  SI_intr^.param[6] := ofs(target);     { pointer to target array  }
  SI_intr^.param[7] := seg(target);
  SI_intr^.param[8] := ofs(option[1]);  { pointer to option string }
  SI_intr^.param[9] := seg(option[1]);
  printfln ('AINSC : opt = ' + option);
  AINSC := RUN_ISAAC;                   { return with error code   }
end;

{.pa}
```



```
function AOUS (chan, dac : integer; var opt) : integer;
var  option : string[255] absolute opt;
begin
  option[length(opvion)+1] := #0;     { convert PASCAL string to C format }
  regs.bx := 8;                       { function num      }
  regs.cx := 5;                       { no of param words }
  SI_intr^.param[0] := chan;
  SI_intr^.param[1] := ofs (dac);
  SI_intr^.param[2] := seg (dac);
  SI_intr^.param[3] := ofs (option[1]);
  SI_intr^.param[4] := seg (option[1]);
  printfln ('AOUS : opt = ' + option);
  AOUS := RUN_ISAAC;
end;


function BEEPFUN (var opt) : integer;
var  option : string[255] absolute opt;
begin
  option[length(option)+1] := #0;     { convert PASCAL string to C format }
  regs.bx := 14;                      { function num      }
  regs.cx := 2;                       { no of param words }
  SI_intr^.param[0] := ofs (option[1]);
  SI_intr^.param[1] := seg (option[1]);
  BEEPFUN := RUN_ISAAC;
end;


function STATS (var target, opt) : integer;
var  option : string[255] absolute opt;
begin
  option[length(option)+1] := #0;     { convert PASCAL string to C format }
  regs.bx := 33;                      { function num      }
  regs.cx := 4;                       { no of param words }
  SI_intr^.param[0] := ofs (target);
  SI_intr^.param[1] := seg (target);
  SI_intr^.param[0] := ofs (option[1]);
  SI_intr^.param[1] := seg (option[1]);
  STATS := RUN_ISAAC;
end;

{.pa}

function ISAAC_version (var version, release) : integer;
var ver : string[5] absolute version;
    rel : string[9] absolute release;
begin
  regs.bx := 43;                      { function num      }
  regs.cx := 4;                       { no of param words }
  SI_intr^.param[0] := ofs (ver[1]);
  SI_intr^.param[1] := seg (ver[1]);
  SI_intr^.param[2] := ofs (rel[1]);
  SI_intr^.param[3] := seg (rel[1]);
  ISAAC_version     := RUN_ISAAC;
  ver[0]            := #4;
  rel[0]            := #8;
end;


type _ISAAC_data_type = array[0..1000] of integer;
     _ISAAC_opt_type  = string[255];

var  _AI : record
             low     : integer;
             high    : integer;
             count   : integer;
             rate    : integer;
             data    : ^_ISAAC_data_type;
             opt     : _ISAAC_opt_type;
             disp    : boolean;
```

```
            Isize   : integer;
            error   : integer;
          end;

var  _AO : record
            chan    : integer;
            dac     : integer;
            opt     : _ISAAC_opt_type;
            error   : integer;
          end;

procedure INIT_ISAAC;
begin
  with _AI do begin
    low  := 0;       high := 0;       count:= 0;       rate := 255;
    opt  := $0;      disp := true;    Isize:= 0;       data := nil;
  end;
  with _AO do begin
    chan := 0;       dac  := 0;         opt  := $0;
  end;
  enable.AI := false;
  enable.AO := false;
end;

{.pa}

procedure expend_opt (var opt : _ISAAC_opt_type; var buf : st128);
var i   : integer;
    num : boolean;
begin
  opt := '';
  i := 1;
  num  := true;
  while ((buf[i] <> ',') and (buf[i] <> ';') and (i <= length(buf)) ) do begin
    if (num) then begin
      if (buf[i] in ['0'..'9']) then begin
        opt := opt + ' ';
        num := false;
      end;
    end
    else begin
      if (buf[i] in ['a'..'z','A'..'Z']) then begin
        opt := opt + ' ';
        num := true;
      end;
    end;
    opt := opt + buf[i];
    i := i + 1;
  end;
end; { procedure expend_opt (var opt : _ISAAC_opt_type; var buf : st128); }

{.pa}

function AINSC_output (title : boolean; index : integer ) : st_type;
var i, j, k : integer;
    Str     : st_type;
    fill    : char;
begin
  with _AI do begin
    if (title) then begin
      Str := ' ';
      fill:= '|';
      for i:=low to high do begin
        if (i < 10) then  Str := Str + fill + '_AI('+ chr(48+i) + ')'
        else              Str := Str+ fill + '_AI('+ chr(55+i) + ')';
        fill:= ' ';
      end;
    end
    else begin
      Str := ' ';
```



```
      j := high - low;
      k := (i + j) * index;
      for i:=k to (k + j) do
        Str := Str + I_str (data^[i], 7);
    end;
    AINSC_output := Str;
  end;
end; { function AINSC_output (title : boolean; index : integer ) : st_type; }


procedure Disp_AINSC_Results;
  Var   i,  ii          : integer;
        ch              : char;
  begin
    if (Not _AI.Disp) then exit;

    Open_LDA_Window;
    ClrScr;
    GotoXY(1,1);
    printf ( S_str('', 'R', W[LDA].Xview+1, $240) );

    GotoXY (1,1);
    printf ( S_str(' Index '+AINSC_output(true,0), 'L', W[LDA].Xview+1, $240) );

    with W[LDA] do Window (Xpos, Ypos+1, Xpos + Xview, Ypos + Yview);
    GotoXY (1,1);
    i := 0;
    for i:=0 to _AI.count-1 do begin
      printfln ( I_str(i, 6) + ' ' + AINSC_output(false,i) );
    end;
    with W[LDA] do Window (Xpos, Ypos, Xpos + Xview, Ypos + Yview);
    Open_USER_Window;
  end; { procedure Disp_AINSC_Results }

{.pa}

procedure cli_AINSC ( token   : st3;          { sub command token }
                      var buf : st128);       { command line      }
var data10            : st10;
    data2             : string[2];
    r_data            : real;
    v_code, err, i, j : integer;
    num               : boolean;
begin
  if (NOT enable.AI) then exit;

  Push_Msg (' Proc: cli_AINSC');
    err := 0;
    data2 := buf;
    with _AI do begin
      if   (token='LOW') or (token='HIG')
        or (token='COU') or (token='RAT') then begin
        if ( FoundData (buf, data10) ) then begin
          VAL (data10, r_data, v_code);
          if (v_code = 0) then begin
            if (token = 'LOW') then begin
              low := trunc (r_data);
              if (low  <   0) then low  := 0;
              if (low  >  15) then low  := 15;
              if (high < low) then high := low;
            end
            else if (token = 'HIG') then begin
              high := trunc (r_data);
              if (high <    0) then high := 0;
              if (high >   15) then high := 15;
              if (low  > high) then low  := high;
            end
            else if (token = 'COU') then begin
              count := trunc (r_data);
              if (count < 0) then count := 0;
```

```
          end
          else if (token = 'RAT') then begin
            rate := trunc (r_data);
            if (rate <  0) then rate := 0;
          end;
        end
        else err := 1;
      end
      else err := 2;
    end
    else if (token='DIS') then begin
      if      (data2 = 'ON') then disp := true
      else if (data2 = 'OF') then disp := false
      else if (data2 = 'AG') then Disp_AINSC_Results;
    end
    else if (token='OPT') then begin
      expend_opt (opt, buf);
    end
    else err := 3;
  end;
Pop_msg;
end; { cli_AINSC }


procedure Read_AINSC;
begin
  with _AI do begin
    if (Isize > 0) then FreeMem (data, Isize);
    Isize := (high - low + 1) * count;
    if (Isize > 0) then begin
      GetMem  (data, Isize);
      error := AINSC (low, high, count, rate, data^[0], opt);
      Disp_AINSC_Results;
    end;
  end;
end; { Read_AINSC }

{.pa}

procedure cli_AOUS  ; token   : st3;              { sub command token }
                      var buf : st128);           { command line      }
var data            : st10;
    data2           : string[2];
    r_data          : real;
    v_code, err, i, j : integer;
    num             : boolean;
begin
  if (NOT enable.AO) then exit;

  Push_Msg (' Proc: cli_AOUS');
    err := 0;
    data2 := buf;
    with _AO do begin
      if   (token='CHA') or (token='DAC') then begin
        if ( FoundData (buf, data) ) then begin
          VAL (data, r_data, v_code);
          if (v_code = 0) then begin
            if (token = 'CHA') then begin
              chan := trunc (r_data);
              if (chan <  0) then chan := 0;
              if (chan > 15) then chan := 15;
            end
            else if (token = 'DAC') then begin
              dac := trunc (r_data);
            end
          end
          else err := 1;
        end
        else err := 2;
      end
```



```
      else if (token='OPT') then begin
        expend_opt (opt, buf);
      end
      else err := 3;
    end;
  Pop_msg;
end; { cli_AOUS }


function AOUS_output (title : boolean ) : st_type;
var Str  : st_type;
begin
  with _AO do begin
    if (title) then begin
      if (chan < 10) then  Str := ' |_AO(' + chr(48+chan) + ')'
      else                 Str := ' |_AO(' + chr(55+chan) + ')';
    end
    else begin
      Str := ' ' + I_str (dac, 7);
    end;

    AOUS_output := Str;
  end;
end; { function AOUS_output (title : boolean ) : st_type; }


procedure Read_AOUS;
begin
  with _AO do begin
    error := AOUS (chan, dac, opt);
  end;
end; { Read_AOUS }
```

```
{ DEFINITION OF GLOBAL CONSTANTS AND VARIABLES }
{=======================================}

const max_lda_array = 10000;

type _LDA_data_type = record
                        Pm  : byte;
                        Px  : byte;
                        F   : byte;
                        Tm  : byte;
                        TDx : byte;
                        Dm  : byte;
                      end;

     _LDA_data_arr1 = array[0..max_lda_array] of _LDA_data_type;

     _LDA_data_arr_type = ^_LDA_data_arr1;

var  _LDA : record
              mode     : byte;          {0 = disp raw data            }
                                        {1 = mode 1: fD, T            }
                                        {2 = mode 2: fD, F, T         }
                                        {3 = mode 3: Bt, IBt, F, T    }
                                        {4 = mode 4: Tt, ITt, T       }
              DispMode : boolean;       {ON  = Disp result after read }
              TimeBase : real;          {S=0 = external time base     }
                                        {S=2 = Tb = 1 msec            }
                                        {S=3 = Tb = 100 msec          }
              S        : integer;
              nsets    : integer;       {n sets of 6 bytes            }
              size     : real;          {size of dma buffer in bytes  }
              Isize    : integer;       {same ar above (in integer format)  }
              SetRead  : integer;       {sets of data read during Read cycle }
              Interval : real;          {delay before taking the next reading}
              DelayFactor : real;       {delay factor determine from calibration}
              dma      : byte;          {dma channel number (1 or 3)  }
              data     : _LDA_data_arr_type;
            end;

const
  base     = $8300;
  dma      = $0;
  dma_page = $80;

var
  data_seg, data_ofs,
  cur_word_count     : integer;

{.pa}

{ DEFINITION OF PROCEDURES }
{==========================}
procedure INIT_LDA_INTERFACE (count : integer; mode : char);

var
  command, offset  : array[0..2] of byte;
  i, counter       : integer;

begin
  command[0] := $89;                { 8255-0 pa,pb=output, pc=input }
  command[1] := $9b;                { 8255-1 pa,pb,pc=input         }
  command[2] := $9b;                { 8255-2 pa,pb,pc=input         }
  offset[0] := 0;                   { offset address of 8255-0      }
  offset[1] := 8;                   { offset address of 8255-1      }
  offset[2] := 12;                  { offset address of 8255-2      }

  port[base + 4] := 0;              { issue a software reset        }

  for i:=0 to 2 do                  { load each 8255 mode register  }
    port[base +offset[i] +3] := command[i];
```



```
                                    { load count to interface card  }
  if (count >0) and (count <7) then begin
    if (count <4) then  counter := 6 - count + 1
    else                counter := 6 - count;
  end
  else begin
    printfln ('');      printfln (bell + 'COUNT ERROR ');
    exit;
  end;

  port[base + 1] := _LDA.S and $03;          { configure the Time Base }

  if (mode = 'D') then begin
    { setup for DMA mode ------------------------------------------}
    { load LDA command reg.    en:DMA dis:INT  en:ARM  en:INH  set:count}
    port[base +offset[0] +0] := $80 or $00 or $20 or $10 or counter;
    port[base +6] := 0;                          { clear inhibit and DMA}
  end
  else begin
    { setup for POLLING mode --------------------------------------}
    { load LDA command reg.   dis:DMA dis:INT dis:ARM dis:INH  set:count}
    port[base +offset[0] +0] := $00 or counter;
    port[base +5] := 0;                          { set internal counter }
    port[base +6] := 0;                          { clear inhibit and DMA}

    { load LDA command reg.   dis:DMA dis:INT  en:ARM  en:INH  set:count}
    port[base +offset[0] +0] := $00 or $00 or $20 or $10 or counter;
  end;
end;

{.pa}

{-------------------------}
procedure INIT_and_RUN_DMA;

VAR  dma_command, temp : integer;

begin
  dma_command := $44 or _LDA.dma; { =0100 01**B
                                  | || | **- channel #
                                  | || **-- write (to memory) transfer
                                  | |*------ disable autoinitialization
                                  | *------- increment address mode
                                  **-------- single byte transfer       }
  port[dma +11] := dma_command;

  port[dma +12] := dma_command;   { reset first/last flip flop          }

  data_seg := seg( _LDA.data^ );          { find segment address of data        }
  data_ofs := ofs( _LDA.data^ );          { find offset address of data         }

  { *** load page_reg with 1st 4 bits of 20 bit addr }
  if (_LDA.dma = 1) then
    port[dma_page +3] := (data_seg and $f000) shr 12
  else begin
    if (_LDA.dma = 3) then
      port[dma_page +2] := (data_seg and $f000) shr 12
    else
      printfln ('DMA channel SELECTION ERROR ');
  end;
  printfln ('dma_page = ' + HEX((data_seg and $f000) shr 12) );

  { *** load dma controller with buffer start addr }
  temp := ((data_seg and $0fff) shl 4) + data_ofs;
  port[dma +(_LDA.dma *2) +0] := lo(temp);
  port[dma +(_LDA.dma *2) +0] := hi(temp);
  printfln ('buffer start address= '+ HEX(((data_seg and $0fff) shl 4) + data_ofs) );
  printfln ('lo= '+ HEX( lo(temp) ) + '   hi= ' + HEX( hi(temp) ) );

  temp := TRUNC(_LDA.size);           { load word count to dma controller     }
```

```
  port[dma +(_LDA.dma *2) +1] := lo(temp);
  port[dma +(_LDA.dma *2) +1] := hi(temp);
  printfln ('# of reading= ' + HEX( temp ) );

  { Unmask and run }
  port[dma + 10] := _LDA.dma;
end;

{.pa}

{----------------------------------------}
procedure WAIT_TILL_COMPLETE (mode : char);

const
  tcp : array[1..3] of byte = (2, 0, 8);  {bit patten of TC on channel 1 and 3}

var
  dma_status,
  i, j              : integer;
  x, y              : integer;
  interrupted       : boolean;
  ByteRead          : real;
  Tdelay            : integer;

begin
  interrupted := false;
  _LDA.SetRead := 0;
  Tdelay := Trunc (_LDA.Interval * 1000.0 * _LDA.DelayFactor);

  printfln ('');

  port[base +5] := 0;           { trigger the ARM pulse         }

  if (mode = 'D') then begin
    { -- wait for DMA to complete -- }
    { by read dma controller's status register to determine end of dma cycle}
    printf ('The current DMA word count is ');
    x := whereX;  y := whereY;
    repeat
      cur_word_count := port[dma +( _LDA.dma *2) +1] + 1;
      cur_word_count := cur_word_count + port[dma +( _LDA.dma *2) +1] shl 8;
      GotoXY (x, y);
      printf ( HEX (cur_word_count) );
      dma_status := port[dma +8] and $0f;
      if (keypressed) then begin
        interrupted := true;
      end;
    until (((dma_status = tcp[_LDA.dma]) or (interrupted)) or (cur_word_count = 0));
  end
  else begin
    { -- wait for POLLING to complete -- }
    printf ('Currently POLLING : ');  x := whereX;  y := whereY;
    printf ('     / of ' + I_str (_LDA.nsets, 5) );
    i := 0;
    repeat
      if ( (port[base +2] and $80) = $80 ) then begin
        GotoXY (x, y);
        printf ( I_str( i+1, 5 ) );
        with _LDA.data^[i] do begin
          Pm := port[base +7];  Px := port[base +7];  F  := port[base +7];
          Tm := port[base +7];  TDx:= port[base +7];  Dm := port[base +7];
        end;
        i := i + 1;
        port[base +6] := 0;           { clear INH line }
        port[base +5] := 0;           { send the next ARM pulse }
      end;

      if (keypressed) then begin
        interrupted := true;
      end
```



```
      else delay (Tdelay);

    until ( (i >= _LDA.nsets) or (interrupted) );
  end;

  port[base + 0] := 0;                { release lda -INH signal }

  { -- calculate how many set of data are read -- }
  if (mode = 'D') then begin
    i := cur_word_count;
    if (i < 0) then ByteRead := 65536.0 - i
    else            ByteRead := i;
    _LDA.SetRead := _LDA.nsets - Trunc (ByteRead / 6);
  end
  else begin
    _LDA.SetRead := 1;
  end;

  printfln ( ');
  if (interrupted) then
    printf (bell + 'PROGRAM HALTED BY USER')
  else
    printf ('DMA sequence completed');
  printfln (' >> ' + I_str (_LDA.SetRead, 5) + ' Sets Read <<');
end;
```

```
(* TURBO Pascal Declarations                                    *)

(*$V-*)          (* relax string length restrictions           *)

Const
      MAXIBBUF = $100;  (* maximum buffer size for I/O functions          *)
      MAXINTBUF = $100; (* maximum buffer size for integer I/O functions *)

Type
       iobuf =  array[1..MAXIBBUF] of char;
       iolbuf = array[1..MAXINTBUF] of integer;
       ibstring = string[50];
       str4 = string[4];
 Var
    ibsta : integer;   (* status word                 *)
    iberr : integer;   (* GPIB error code             *)
    ibcnt : integer;   (* number of bytes sent or, in the event of *)
                       (* DOS error, the DOS error code           *)
    ibbuf : iobuf;     (* I/O buffer for commands/data   *)
    intbuf : iolbuf;   (* integer I/O buffer for data    *)
    bdname : ibstring; (* board or device name           *)
    bd,dvm : integer;  (* Board descriptor               *)
    vcnt: integer;     (* v or byte count                *)
    flname : ibstring; (* file name                      *)
    mask : integer;    (* wait mask for IBWAIT ftn.      *)
    ppr,spr:integer;   (* parallel,serial poll responses *)


(* Ibfn is the common entry point into the language interface, tpib.com.   *)
(* Its arguments are generalized to meet the needs of each individual GPIP *)
(* function, and are decoded as follows:                                   *)
(*          ibfnasm (name,iberr,ibcnt,buf,ibuf,vcnt,bd,fcode,iberr,ibcnt)  *)
(* where:       name  = string for bdname, flname, bname                   *)
(*              iberr = GPIB-PC error code                                 *)
(*              ibcnt = GPIB-PC count                                      *)
(*              ibuf  = integer array for (var) rdi, wrti buffers          *)
(*              buf   = integer array for (var) rd, wrt, cmd buffers       *)
(*              vcnt  = integer for v, cnt, (var) spr, (var) ppr           *)
(*              bd    = integer for bd                                     *)
(*              fcode = integer for function code                          *)


function ibfn (name:ibstring;var iberr,ibcnt:integer;var buf:iobuf;var bufl:iolbuf;
     var vcnt:integer;bd,fncode:integer):integer; external 'dd-NI488.bin';


(* You MUST include the appropriate declaration, as            *)
(* given below, for each procedure or function you call:       *)
(* You may omit declarations for functions you do not call.    *)

procedure    ibbna (bd:integer;bname:ibstring);
    var
    name : ibstring;
    begin
    name := bname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,26);
    end;
procedure    ibcac (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,16);
    end;
procedure    ibclr (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,22);
    end;
procedure    ibcmd (bd:integer;ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,32);
```



```
    end;
procedure    ibcmda (bd:integer;ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,33);
    end;
procedure    ibdiag (bd:integer;ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,34);
    end;
procedure    ibdma (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,11);
    end;
procedure    ibeos (bd:integer;v:integer);
    begin
ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,12);
    end;
procedure    ibeot (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,14);
    end;
function     ibfind (bdname:ibstring):integer;
    var
    name : ibstring;
    begin
    name := bdname + chr(0);
    ibfind := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,27);
    end;
procedure    ibgts (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,15);
    end;
procedure    ibist (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,10);
    end;
procedure    ibloc (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,5);
    end;
procedure    ibonl (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,1);
    end;
procedure    ibpad (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,8);
    end;
procedure    ibpct (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,24);
    end;
procedure    ibppc (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,7);
    end;
procedure    ibrd (bd:integer;var ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,28);
    end;
procedure    ibrdi (bd:integer;var intbuf:iolbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,36);
    end;
procedure    ibrda (bd:integer;var ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,29);
    end;
procedure    ibrdf (bd:integer;flname:ibstring);
    var
```

```
    name : ibstring;
    begin
    name := fname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,17);
    end;
procedure    ibrpp (bd:integer;var ppr:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,ppr,bd,19);
    end;
procedure    ibrsc (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,2);
    end;
procedure    ibrsp (bd:integer;var spr:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,spr,bd,25);
    end;
procedure    ibrsv (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,6);
    end;
procedure    ibsad (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,9);
    end;
procedure    ibsic (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,3);
    end;
procedure    ibsre (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,4);
    end;
procedure    ibstop (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,21);
    end;
procedure    ibtmo (bd:integer;v:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,v,bd,13);
    end;
procedure    ibtrg (bd:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,23);
    end;
procedure    ibwait (bd:integer;mask:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,mask,bd,0);
    end;
procedure    ibwrt (bd:integer;ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,30);
    end;
procedure    ibwrti (bd:integer;intbuf:iolbuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,37);
    end;
procedure    ibwrta (bd:integer;ibbuf:iobuf;cnt:integer);
    begin
    ibsta := ibfn(bdname,iberr,ibcnt,ibbuf,intbuf,cnt,bd,31);
    end;
procedure    ibwrtf (bd:integer;fname:ibstring);
    var
    name : ibstring;
    begin
    name := fname + chr(0);
    ibsta := ibfn(name,iberr,ibcnt,ibbuf,intbuf,vcnt,bd,18);
    end;
```

```
{-------------------------------------------------------------------}
{ DISA TRAVER Driver Routine                                        }
{ -------------------------                                        }
{  1) Init_Traver                        { Reset calibration constant }
{     TRAVER  (com         :trav_dev_cpt;{ I : read, set or clear     }
{              channel_num :byte;        { I : channel num <0..2>     }
{              Var position:real);       {I/O: real in mm             }
{-------------------------------------------------------------------}

Type   trav_dev_cpt  = dev_cpt;

Var    trav_id       : integer;
       trav_err_msg  : st128;

TYPE Array3          = array [0..2] of byte;

VAR  pulse_per_mm    : array [0..3] of real;


Procedure Init_TRAV_DRV;
   BEGIN
      enable.trav  := FALSE;

      pulse_per_mm[0] := 240.0;
      pulse_per_mm[1] := 240.0;
      pulse_per_mm[2] := 240.0;
      pulse_per_mm[3] := 240.0;
  END;
{.pa}

{------------- TRAVER Driver ---------------------------------------}
type trav_type = (data_c, position_c);

Procedure CONVERT_TO (com : trav_type;      { I : data or position       }
                      ch_num : byte;        { I : channel number         }
                      Var data: array3;     {I/O: 2's comp of pos. 24 bit}
                      Var position :real); {I/O: real in mm              }

  Var data_r         : array [0..2] of real;
      posn           : real;
      i, sign_code   : byte;

  BEGIN
    Push_msg (' Proc: CONVERT_TO');
      If (com = data_c) then begin       { convert to binary DATA }
        posn := int (position * pulse_per_mm[ch_num]);

        If (posn < 0.0) then begin
          sign_code := $FF;
          posn := abs( posn) -1;
        end
        Else
          sign_code := $00;

        If ((posn < 8388608.0) and (posn  >= 0)) then begin
          data_r[0] := posn / 65536.0;                 { MSB    [0] }
          data_r[1] := frac( data_r[0] ) * 256.0;      { MSB-1  [1] }
          data_r[2] := frac( data_r[1] ) * 256.0;      { LSB    [2] }
          For i:=0 to 2 do begin
            data[i] := trunc( data_r[i] ) xor sign_code;
          end;
        end
        Else begin
          Push_msg ('  ** invalid POSITION entry');
          Abort;
        end;
      end
      Else begin
        If (com = position_c) then begin     { convert to POSITION }
          If ((data[0] and $80) >0) then begin
```



```
            position := -1; sign_code := $FF;
          end
          Else begin
            position :=  1; sign_code := $00;
          end;
          position := position * (    ( data[0] xor sign_code ) * 65536.0
                                   + ( data[1] xor sign_code ) * 256.0
                                   + ( data[2] xor sign_code ) * 1.0 );
          position := position / pulse_per_mm[ch_num];
        end
        Else begin
          Push_msg ('  ** invalid COMMAND **');
          Abort;
        end;
      end;
    pop_msg;
  end; { Proc: CONVERT_TO }
{.pa}

procedure TRAVER  (com :       :trav_dev_cpt; { I : read_d, set_d or clear_d}
                   channel_num :byte;         { I : channel num <0..2>      }
                   Var position:real);        {I/O: real in mm              }

  Const ch_base  = $860;
        no_com   = $00;
        clear_com= $10;    load_com = $20;
        run_com  = $40;    edir_com = $40;

  Var   data     : array3;
        data16   : array16;
        ch_num, i: integer;
        dum      : boolean;

  BEGIN
    If (not Enable.trav) then
      EXIT;
    Push_msg (' Proc: TRAVER');
      ch_num := ch_base + ((channel_num and $03) shl 7);
      If (com = read_d) then begin       { load data to out reg. }
        write_pc (trav_id, $30, ch_num+6, load_com + run_com);
        write_pc (trav_id, $0B, ch_num+6, no_com);   { request 16 byte of data}
        if (read_pc  (trav_id, 16, data16, trav_err_msg)) then
          printfln (bell + 'TRAV unit: ' + trav_err_msg);
        data[0] := data16[2];
        data[1] := data16[4];
        data[2] := data16[6];
        convert_to (position_c, channel_num, data, position);
      end
      Else begin
        If (com = set_d) then begin
          convert_to (data_c, channel_num, data, position);
          write_pc (trav_id, $30, ch_num+0, data[2]);
          write_pc (trav_id, $30, ch_num+2, data[1]);
          write_pc (trav_id, $30, ch_num+4, data[0]);
          write_pc (trav_id, $30, ch_num+6, run_com);
        end
        Else begin
          If (com = clear_d) then begin
            write_pc (trav_id, $30, ch_num+6, clear_com); { clear counter }
            write_pc (trav_id, $30, ch_num+0, 0);         { set coord. to }
            write_pc (trav_id, $30, ch_num+2, 0);         { zero          }
            write_pc (trav_id, $30, ch_num+4, 0);
            write_pc (trav_id, $30, ch_num+6, run_com);
          end
          Else begin
            disp_msg_stack;
            println ('Invalid (TRAVER) command');
            printf ('');
          end;
        end;
```

```
    end;
  Pop_msg;
END; { Proc: TRAVER }
{.pa}
```



```
{-------------------------------------------------------------------}
{ Error Stack Routine :                                             }
{-------------------------------------------------------------------}

Type   st40    = string[40];
       st128   = string[128];
       msg_ptr = ^msg_rec;
       msg_rec = record
                   msg  : st40;
                   next : msg_ptr;
                   prev : msg_ptr;
                 end;

Var    heaptop  : ^integer;
       first_msg, last_msg, new_msg : msg_ptr;

{-------------------------------------------------------------------}
{ Convert an Integer to a Hex string                                }
{-------------------------------------------------------------------}
type hex_type = string[10];

function HEX( DATA : INTEGER): hex_type;
  CONST
    HEX_STR    : ARRAY [0..15] OF CHAR
                   = ('0','1','2','3','4','5','6','7',
                      '8','9','A','B','C','D','E','F');
  BEGIN
    HEX := '$' + HEX_STR[ (DATA AND $F000) SHR 12 ]
               + HEX_STR[ (DATA AND $0F00) SHR  8 ]
               + HEX_STR[ (DATA AND $00F0) SHR  4 ]
               + HEX_STR[ (DATA AND $000F)        ];
  END; { Proc: HEX }
{-------------------------------------------------------------------}
{.pa}

{-------------------------------------------------------------------}
{ Error Stack Routine :                                             }
{ --------------------                                              }
{   1)  Init_Msg_Stack;              * Get pointer to Heap          }
{                                                                   }
{   2)  Push_Msg (Err_Msg);          * Push to message stack        }
{                                                                   }
{   3)  Pop_Msg;                     * Remove message from stack    }
{                                                                   }
{   4)  Disp_Msg_Stack;              * Non-distractive disp of stack }
{                                                                   }
{   5)  Abort;                       * Display msg stack & Halt prog }
{-------------------------------------------------------------------}

{ ==== initialize the error message stack }
Procedure INIT_MSG_STACK;
  BEGIN
    first_msg := nil;
    last_msg  := nil;
  END; { Proc: INIT_MSG_STACK }


{ ==== push error message to a dynamic stack }
Procedure PUSH_MSG ( ERR_MSG : st40 );
  Var  temp_ptr : msg_ptr;

  BEGIN
    new (new_msg);
    new_msg^.msg := err_msg;
    If (first_msg = nil) then begin
      first_msg := new_msg;
      temp_ptr  := nil;
    end
    Else begin
      last_msg^.next := new_msg;
```

```
      temp_ptr  := last_msg;
    end;
    last_msg := new_msg;
    last_msg^.next := nil;
    last_msg^.prev := temp_ptr;
  END; { Proc: PUSH_MSG }
{.pa}

{ ==== pop the error message when the procedure exec. correctly }
Procedure POP_MSG;
  Var temp_ptr : msg_ptr;

  BEGIN
    If (last_msg = nil) then begin
      Writeln ('  ** Unmatched POP_MSG error occur **');
      Halt;
    end
    Else begin
      If (last_msg = first_msg) then begin
        DISPOSE (last_msg);
        first_msg := nil;
        last_msg := nil;
      end
      Else begin
        temp_ptr := last_msg^.prev;
        temp_ptr^.next := nil;
        temp_ptr := last_msg;
        last_msg := last_msg^.prev;
        DISPOSE (temp_ptr);
      end;
    end;
  END; { Proc: POP_MSG }

{ ==== non-distructive display of the error message stack }
Procedure DISP_MSG_STACK;
  Var  Temp_ptr : Msg_ptr;

  BEGIN
    Writeln;
    Writeln ('Non-Districtive Display Of Massage Stack');
    Writeln ('-------------------------------------');
    temp_ptr := first_msg;
    While (temp_ptr <> nil) do begin
      writeln (temp_ptr^.msg);
      temp_ptr := temp_ptr^.next;
    end;
  END; { Proc: DISP_MSG_STACK }
{.pa}

{ ==== ABORT the program and display the error message }
Procedure ABORT;
  BEGIN
    Writeln ;
    Writeln ('          ** Error Condition Occur **');
    Writeln ('A list of error messages from the most top level down');
    Writeln ('-----------------------------------------------------');
    While (first_msg <> nil) do begin
      Writeln (first_msg^.msg);
      first_msg := first_msg^.next;
    end;
    Halt;
  END; { Proc: ABORT }
{.pa}
```



```
{ Written by  : Alex Tsui   }
{ Last Update :  1/26/87    }

procedure clean_input_cmd (var buf : st128);
  var i      : byte;
      token3 : st3;
  begin
    i := pos (';', buf);
    DELETE (buf, i+1, LENGTH(buf)-i);           { delete everything after ';'}

    if (i>1) then begin
      while (buf[1] = ' ') do delete (buf, 1, 1); { delete leading blanks    }

      FOR i:=1 to 3 DO begin                      { convert text to upcase   }
        buf[i]    := UPCASE (buf[i]);
      end;
      token3 := buf;

      if (token3 <> 'DOS') then begin
        DeleteCHAR ('*', buf);                    { remove all '*' & ' ' char}
        DeleteChar (' ', buf);
        FOR i:=1 to LENGTH(buf) DO                { convert text to upcase   }
          buf[i] := UPCASE (buf[i]);
      end;
    end;
  end; { clean_input_cmd }

{.pa}

Procedure Get_Input (msg : st10;
                     Var buf : st128);
  VAR  x, y, i              : integer;
       key_press            : boolean;
       temp_buf             : st128;
       start_str, cont_str : st80;
       token3               : st3;
       ch                   : char;

  procedure store_input (cont_prompt : st80);
    begin
      buf := buf + temp_buf;
      if ( length(buf) > 125 ) then begin
        buf := ';';
        printfln ('');
        printfln (bell + '...Line too long, text ignored...');
      end
      else begin
        i := pos (';', buf);
        if (i = 0) then begin
          if (temp_buf = '') then begin
            buf := buf + ';';
            i := pos (';', buf);
            printf (';');
          end
          else begin
            printfln ('');
            printf   (cont_prompt);
          end;
        end;
      end;
    end;

  BEGIN
     buf := '';
     IF (cmd_num >= 0) then begin
        IF (EOF(cmd_file[cmd_num])) then begin
           CLOSE (cmd_file[cmd_num]);
           cmd_file_exist[cmd_num] := FALSE;
           IF (cmd_num >= 0) then
              cmd_num := cmd_num - 1;
```

```
        end
        ELSE begin
          Str (cmd_num:1, start_str);
          start_str := start_str + msg;
          start_str := 'FILE:' + start_str;
          cont_str  := start_str + ' ';
          start_str := start_str + ': ';
          printfln (''); printf (start_str);
          repeat
            READLN (cmd_file[cmd_num], temp_buf);
            printf (temp_buf);
            delay  (200);
            Store_input (cont_str);
          until (i>0);
        end
      end

      ELSE begin
        printfln ('');
        printf   ('CMD' + msg + '> ');
        TextColor     (W[input].foreground + blink);
        TextBackGround(W[input].background); printf(#16);
        x := WhereX;  y := WHEREY;
        key_press := false;
        REPEAT
          Update_Time_Date (false);
        UNTIL (keypressed);
        Read (KBD, ch);
        if (ch = #0) then begin
          Read (KBD, ch);
          if (ch = #59) then            {F1}
            buf := 'HELP;'
          else if (ch = #60) then       {F2}
            buf := 'WINDOW;';
          Open_CMD_window;
          GOTOXY (x-1, y);
          printf (buf);
        end
        else if (ch = #13) then begin  {cr}
          buf := ';';
          Open_CMD_window;
          GOTOXY (x-1, y);
          printf (buf);
        end
        else begin
          Open_CMD_window;
          GOTOXY (x-1, y);
          printf (ch);
          buf := ch;
          Repeat
              READ (temp_buf);
              Enable_cursor;
              Store_input ( msg + '**      ');
          until (i>0);
        end;
      end;
      clean_input_cmd (buf);
END; { Get_Input }
```



```
{ - - Frequently used procedures & functions - - - - - - - - - - - - }

Procedure DeleteCHAR (ch : char;
                      Var Buf : st128); {I/O: contain string t.b.c.  }
  Var  i    : byte;
  BEGIN
     REPEAT
        i := POS (ch, buf);
        IF (i > 0) THEN
           DELETE (buf, i, 1);
     UNTIL (i=0);
  END;
{.pa}

Function  FoundCom (VAR buf     : st128; {I/O: line buffer             }
                    delim       : char;  { I : delimeter               }
                    Var com_buf : st10)  { O : return command string   }
                    : BOOLEAN;           { Return TRUE = command found}

  VAR i     : BYTE;
      found : BOOLEAN;
  BEGIN
     com_buf := '';
     i := POS (delim, buf);
     IF (i > 0) THEN BEGIN
        com_buf := COPY (buf, 1, i-1);
        DELETE (buf, 1, i);
        found := TRUE;
     END
     ELSE
        found := FALSE;
     FoundCom := found;
  END; { FoundCom }
{.pa}

Function FoundData (VAR buf      : st128; {I/O: line buffer             }
                    Var data_buf : st10)  { O : return data string     }
                    : BOOLEAN;            { Return TRUE if string found}

  Var found : BOOLEAN;
  BEGIN
     found := FoundCom (buf, ',', data_buf);
     IF (NOT found) THEN
        found := FoundCom (buf, ';', data_buf);
     FoundData := found;
  END; { FoundData }
{.pa}

{ -- CURSOR routines -- }
procedure SetCrtMode (CrtMode : integer);
  begin
    regs.ax := $00FF and CrtMode;
    Intr($10, regs);
  end;

procedure SetCurType (start_line, end_line : byte);
  begin
    regs.ah := $01;
    regs.ch := start_line;
    regs.cl := end_line;
    Intr($10, regs);
  end;

procedure Enable_cursor;
  begin
    SetCurType     (6, 7);                    { enable cursor underscore }
  end;

{* ------------------------------------------------------------------------ *}
procedure Hide_cursor;
```

```
  begin
    SetCurType     ($20, 15);                    { hide cursor }
  end;

function GetVideoMode : integer;
  begin
    Intr($11, regs);
    GetVideoMode := regs.ax;
  end;

procedure GetCursor (Page                    : byte;
                     Var Row, Col, CurMode : byte);
  begin
    with regs do begin
      ax := $0300;
      bx := Page;
      Intr($10, regs);
      Row         := dh;
      Col         := dl;
      CurMode     := cx;
    end;
  end;

procedure SetCursor (Page, Row, Col : byte);
  begin
    with regs do begin
      ah := 2;
      bh := Page;
      dh := Row;
      dl := Col;
      Intr($10, regs);
    end;
  end;

type SendStrType = string[128];

procedure SendStr (charStr : SendStrType);
  var i : byte;
  begin
    with regs do begin
      ah := $06;
      for i:=1 to length(charStr) do begin
        dl := integer(charStr[i]);
        msdos (regs);
      end;
    end;
  end;
** ------------------------------------------------------------------------ *)

{ -- TIME and DATE routines -- }
Type time_date_string = string[6];

function Get_time : time_date_string;
  var  time_str  : time_date_string;
       i         : byte;
       buf2      : string[2];

  BEGIN
     WITH regs DO begin
        ax := $2C00;
        MSDOS (regs);
        STR (HI(cx):2, buf2);   time_str := buf2 + ':';           { hour }
        STR (LO(cx):2, buf2);   time_str := time_str + buf2 + ':'; { min  }
        STR (HI(dx):2, buf2);   time_str := time_str + buf2;       { sec  }
        FOR i:=1 to 8 do begin
          IF (time_str[i] = ' ') then   time_str[i] := '0';
        end;
        Get_time := time_str;
     end;
  END;
```



```
{.pa}

function Get_Seconds : real;
  BEGIN
     WITH regs DO begin
        ah := $2C;
        MSDOS (regs);
        Get_Seconds := (((ch * 60.0) + cl)* 60.0) + dh + (dl / 100.0);
     end;
  END;
{.pa}

function Get_DATE : time_date_string;
  var  date_str  : time_date_string;
       buf4      : string[4];
       i         : byte;
  BEGIN
     WITH regs DO begin
        ax := $2A00;
        MSDOS (regs);
        STR (cx:4,      buf4);   date_str := buf4[3] + buf4[4] + '/'; { year }
        STR (HI(dx):2, buf4);    date_str := date_str + buf4 + '/';   { month}
        STR (LO(dx):2, buf4);    date_str := date_str + buf4;         { day  }
        FOR i:=1 to 8 do begin
          IF (date_str[i] = ' ') then   date_str[i] := '0';
        end;
        Get_Date := date_str;
     end;
  END;
{.pa}
```

```
procedure RAM_Page( pageHandle : integer);          external 'ram-page.bin';

procedure Create_Page( var pageHandle : integer;
                           width,
                           height     : integer;
                       var result     : integer);   external Ram_Page[3];

procedure View_Page(     pageHandle: integer;
                     var x1,
                         y1,
                         x2,
                         y2 : integer);             external Ram_Page[6];

procedure Position_Page(     pageHandle : integer;
                         var xPos       : integer;
                         var yPos       : integer); external Ram_Page[9];

procedure Appear_View( pageHandle : integer);       external Ram_Page[12];

procedure Disappear_View( pageHandle : integer);    external Ram_Page[15];

procedure Frame_View( pageHandle  : integer;
                      frameStyle,
                      attribute   : byte);          external Ram_Page[18];

procedure Init_Ram_Page;                            external Ram_Page[21];

procedure Disable_View( pageHandle : integer);      external Ram_Page[24];

procedure Enable_View( pageHandle  : integer);      external Ram_Page[27];

procedure Copy_Window( pageHandle,
                       xPos,
                       yPos : integer);             external Ram_Page[30];
```



```
{-------------------------------------------------------------}
{ Written by   : Alex Tsui                                    }
{ first version: Sept 15, 86                                  }
{ Last Update  : 18:15:43  2/22/1987                          }
{-------------------------------------------------------------}

const   k_up    = 18432;  s_up    = 56;
        k_down  = 20480;  s_down  = 50;
        k_left  = 19200;  s_left  = 52;
        k_right = 19712;  s_right = 54;

        k_home  = 18176;
        k_end   = 20224;
        k_PgUp  = 18688;
        k_PgDn  = 20736;

        k_f1 = 15104;  k_f2 = 15360;  k_f3 = 15616;  k_f4 = 15872;
        k_f5 = 16128;  k_f6 = 16384;  k_f7 = 16640;  k_f8 = 16896;
        k_f9 = 17152;  k_f10= 17408;

type    RamPageRec = record
          name                      : string[20];
          handle,
          width, height,
          x1, y1, x2, y2,
          xView, yView, xOfs, yOfs,
          foreGround, backGround,
          xPos, yPos                : integer;
          frame_byte                : integer;
        end;

        Windows_Rec = Array[Windows] of RamPageRec;

var     Wresult                     : integer;
        W                           : Windows_Rec;
        i, ii                       : integer;

var     W_recs : array[Windows] of Window_Limits_Rec absolute W;

const   ChgScr : RamPageRec
               = ( name  : 'Change Screen';
                   handle: 0;
                   width : 80;   height:  2;
                   x1    : 1;    y1    : 24;   x2  : 80;  y2  : 26;
                   xView : 0;    yView :  0;   xOfs:  0;  yOfs:  0;
                   foreGround : black;         backGround : white;
                   xPos : 1;     yPos  :  1;   frame_byte : 0 );

{.pa}

procedure Init_Window_Limits;
  var setupFile : text;
      i         : byte;

  begin
    Assign (setupFile, 'cada-lda.def');
    {$I-} Reset (setupFile);  {$I+}
    if (IOresult = 0) then
      for i:=0 to num_of_windows-1 do begin
        with W[Windows(i)] do begin
          readln (setupFile, name);
          readln (setupFile, width, height);
          readln (setupFile, x1,    y1,    x2,   y2);
          readln (setupFile, xView, yView, xOfs, yOfs);
          readln (setupFile, foreground, background);
          readln (setupFile, xPos,  yPos);
          readln (setupFile, frame_byte);
        end;
      end
    else
```

```
      W_recs := Window_Limits;
    close (setupFile);
  end;

type  screen_type = record
                      ch    : char;
                      attr  : byte;
                    end;
var   screen      : array[1..25, 1..80] of screen_type absolute $b800:$0000;

procedure Appear_Window (NameW : RamPageRec);
var  len,i : integer;

begin
  with NameW do begin
    appear_view (NameW.handle);

    if (frame_byte <> 0) then begin
      len   := length(name);

      if ((Xview > 2) and (len > 0)) then begin
        if (len > Xview-2) then
          len := Xview - 2;
        for i:=1 to len do
          screen[Y1-1, X1+i].ch := name[i];
      end;
    end;
  end;
end;

{.pa}

procedure Init_Window (var  NameW : RamPageRec;
                       appear     : boolean;
                       frame      : boolean);
  begin
    with NameW do begin
        Create_Page (handle, width, height, Wresult);
        if Wresult <> 0 then begin
            writeln ('Create_Page failed, result = ', Wresult);
            halt;
        end;

        xView := x2 - x1;     xOfs := width  - xView;
        yView := y2 - y1;     yOfs := height - yView;

        TextColor (foreGround);
        TextBackGround (backGround);

        ram_page (handle);
        clrscr;
        view_page  (handle, x1, y1, x2, y2);
        position_page (handle, xpos, ypos);
        if frame  then begin
            frame_byte  := $ff;
            frame_view (handle,  2,  $2C);  { $2C => FG:light red, BG:green }
        end
        else frame_byte := $00;
        if appear then
            Appear_Window (NameW);
    end;
  end;

procedure Init_chg_scr;
  begin
    Init_Window (ChgScr, false, false);
  end;
```



```
{.pa}

function Key( var inputValue: integer;
              var funcKey : boolean ): boolean;
  var regpack:  record
                  ax,bx,cx,dx,bp,si,di,ds,es,flags: integer;
                end;
      regByte:  array[ 1..20] of byte absolute regpack;

  begin
    with regpack do begin
      ax := 6 shl 8;
      dx := 255;
      MsDos(regpack);
      if regByte[ 19] and $40 = 0 then begin
        key        := true;
        FuncKey    := false;
        inputValue := regByte[ 1];              { no function key }
        if inputValue = 0 then begin
          funckey := true;
          ax := 6 shl 8;
          dx := 255;
          MsDos(regpack);
          inputValue := regByte[ 1] shl 8;      { function key    }
        end;
      end
      else Key := false;
    end;
  end;


function InKey( var inputValue: integer ) : boolean;
  var funcKey : boolean;
  begin
    while (Not (Key(inputValue, funcKey) )) do;    { wait for keypressed }
    InKey := funcKey;
  end;

{.pa}

type  chg_opts = (All, Scroll, Size, Move, Change, Save, Recall, Delete);

procedure Change_screen (NameW : Windows; opt : chg_opts);
  var FuncKey, more_chg, quit_chg                  : boolean;
      keyValue                                     : integer;
  procedure Check_scroll_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : ypos := ypos - 1;
          k_down  : ypos := ypos + 1;
          k_left  : xpos := xpos - 1;
          k_right : xpos := xpos + 1;
          k_PgUp  : ypos := ypos - yView + 1;
          k_PgDn  : ypos := ypos + yView - 1;
          k_home  : begin
                      xpos := 1;  ypos := 1;
                    end;
          k_end   : begin
                      xpos := 1;  ypos := height;
                    end;
        end;
      end;
    end;

  procedure Check_enlarge_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : y1 := y1 - 1;
```

```
          k_down  : y2 := y2 + 1;
          k_left  : x1 := x1 - 1;
          k_right : x2 := x2 + 1;
        end;
      end;
    end;

  procedure Check_reduce_key;
    begin
      with W[NameW] do begin
        case keyValue of
          s_up    : y2 := y2 - 1;
          s_down  : y1 := y1 + 1;
          s_left  : x2 := x2 - 1;
          s_right : x1 := x1 + 1;
        end;
      end;
    end;

  procedure Check_move_key;
    begin
      with W[NameW] do begin
        case keyValue of
          k_up    : begin  y1 := y1 - 1;  y2 := y2 - 1;  end;
          k_down  : begin  y1 := y1 + 1;  y2 := y2 + 1;  end;
          k_left  : begin  x1 := x1 - 1;  x2 := x2 - 1;  end;
          k_right : begin  x1 := x1 + 1;  x2 := x2 + 1;  end;
        end;
      end;
    end;

  procedure Check_Margin (Var x : integer; min, max : integer);
    begin
      if (x < min) then x := min;
      if (x > max) then x := max;
    end;
{.pa}

  procedure Save_Recall_Setup (opt : chg_opts);
    var setupFile  : text;
        inputOK    : boolean;
        kVal,i     : integer;
    begin
      if (opt = All) then begin
        printf ('F1-Save/ F2-Recall/ F3-Delete WINDOW SETUP FROM DISK');
        gotoXY (71,2);
        printf (' F10=EXIT');
        repeat
          FuncKey := InKey(keyValue);
        until (FuncKey and (keyValue >= k_F1) and (keyValue <= k_F10));
      end
      else begin
        case opt of
          Save   : keyValue := k_F1;
          Recall : keyValue := k_F2;
          Delete : keyValue := k_F3;
        end;
      end;
      gotoxy(1,2); delLine;
      Assign (setupFile, 'window.def');
      case keyValue of
        k_F1 : begin
                 Rewrite (setupFile);
                 for i:=0 to num_of_windows-1 do begin
                   with W[Windows[i]] do begin
                     writeln (setupFile, name);
                     writeln (setupFile, width:10, height:10);
                     writeln (setupFile, x1:10,    y1:10,    x2:10,   y2:10);
                     writeln (setupFile, xView:10, yView:10, xOfs:10, yOfs:10);
                     writeln (setupFile, foreground:10, background:10);
```



```
                     writeln (setupFile, xPos:10,  yPos:10);
                     writeln (setupFile, frame_byte:10);
                   end;
                 end;
                 Close   (setupFile);
                 printf  ('SETUP SAVED :');
               end;
        k_F2 : begin
                 {$I-} Reset (setupFile); {$I+}
                 inputOK := (IoResult = 0);
                 if not inputOK then begin
                   gotoxy (1,2); delLine;
                   printf ('Setup file not found');
                 end
                 else begin
                   printf ('LOADING WINDOW SETUP');
                   for i:=0 to num_of_windows-1 do begin
                     with W[Windows[i]] do begin
                       readln (setupFile, name);
                       readln (setupFile, width, height);
                       readln (setupFile, x1,    y1,    x2,   y2);
                       readln (setupFile, xView, yView, xOfs, yOfs);
                       readln (setupFile, foreground, background);
                       readln (setupFile, xPos,  yPos);
                       readln (setupFile, frame_byte );
                       position_page (handle, xpos, ypos);
                       view_page     (handle, x1, y1, x2, y2);
                     end;
                   end;
                   Close  (setupFile);
                   Appear_Window (W[NameW]);
                 end;
               end;
        k_F3 : begin
                 printf   ('Please press ''Y'' to confirm the deletion');
                 If ( (not (Inkey (kVal))) and (kVal = integer('Y')) ) then begin
                   printfln ('');
                   printf   ('Deleting ...');
                   close    (setupFile);
                   erase    (setupFile);
                 end
                 else begin
                   printfln ('');
                   printf   ('Deleting Aborted');
                 end;
               end;
      end;
    end;


  procedure Change_Frame (background_colnr, frameStyle, attr : byte);
    begin
      with W[NameW] do begin
        TextBackGround (background_color);
        disable_view   (handle);
        disappear_view (handle);
        Frame_View     (handle, frameStyle, attr);
        enable_view    (handle);
        Appear_Window  (W[NameW]);
      end;
    end;
{.pa}

  var scroll_scr, size_scr,
      setup_scr,  move_scr, change_scr            : boolean;
      win_ofs                                     : integer;

  begin
    win_ofs := ord (NameW);
    Change_Frame   (red, 1, 0);
```

```
repeat
  with W[NameW] do begin
    scroll_scr := false;  move_scr   := false;
    size_scr   := false;  change_scr:= false;  setup_scr := false;
    quit_chg   := false;
    Appear_Window  (ChgScr);
    ram_page       (ChgScr.handle);
    TextColor      (W[time_window].foreground);
    TextBackGround (W[time_window].background);
    gotoxy( 1,1);  printf ('[illegible]' +
                           '[illegible]');

    TextColor      (ChgScr.foreground);
    TextBackGround (ChgScr.background);
    gotoxy(20,1);  write (' Currently Editting : ',Name:18, ' ');
    if (opt = All) then begin
      gotoxy( 1,2);  write   (' F5 = Scroll,',
                              ' F6 = Size,',
                              ' F7 = Move,',
                              ' F8 = Next Window,',
                              ' F9 = Setup,      ');

      gotoXY (71,2);
      printf (' F10=EXIT');
      repeat
        FuncKey := InKey(keyValue);
      until (FuncKey and (keyValue >= k_F1) and (keyValue <= k_F10));
    end
    else begin
      case opt of
        Scroll : keyValue := k_F5;
        Size   : keyValue := k_F6;
        Move   : keyValue := k_F7;
        Change : keyValue := k_F8;
        Save   : keyValue := k_F9;
        Recall : keyValue := k_F9;
        Delete : keyValue := k_F9;
      end;
    end;

    gotoxy(1,2); DelLine;
    case keyValue of
      k_F5 : begin
               scroll_scr := true;
               printf('SCROLL SCREEN : use Arrow, Home, End, PgUp, PgDn keys to scroll page'
);
             end;
      k_F6 : begin
               size_scr:= true;
               printf('SIZING WINDOW: use Arrow-ENLARGE / Shift_Arrow-REDUCE');
             end;
      k_F7 : begin
               move_scr   := true;
               printf('MOVE SCREEN : use arrow keys to move window');
             end;
      k_F8 : begin
               Change_Frame (background, 2, $2C);
               change_scr := true;
               win_ofs := win_ofs + 1;
               if (win_ofs >= num_of_windows) then begin
                 win_ofs := 0;
                 NameW := Windows(0)
               end
               else
                 NameW := Windows(win_ofs);
               Change_Frame (red, 1, 0);
             end;
      k_F9 : begin
               setup_scr := true;
               Save_Recall_Setup (opt);
               View_page     (handle,x1,y1,x2,y2);
```



```
               Position_page(handle,xpos,ypos);
               Frame_view    (handle,  2,  $2C);  { $2C => FG:light red, BG:green }
               Appear_Window(W[NameW]);
             end;
      k_F10: begin
               quit_chg   := true;
               disappear_view (ChgScr.handle);
               Change_Frame   (background, 2, $2C);
             end;
    end;
    gotoXY (71,2);
    printf (' F10=EXIT');

    if (not (change_scr) and not (setup_scr)) then begin
      if not (quit_chg) then begin
        Appear_Window     (W[NameW]);
        more_chg := true;
      end;

      while ((not quit_chg) and more_chg) do begin
        FuncKey := InKey(keyValue);
        if (keyValue = k_F10) then
          more_chg := false
        else begin
          if      (scroll_scr) then check_scroll_key
          else if (size_scr)   then begin
                                      check_enlarge_key;
                                      check_reduce_key;
                                    end
          else if (move_scr)   then check_move_key;

          Check_Margin (x1,  1, 79);
          Check_Margin (x2, x1, 80);

          Check_Margin (y1,  1, 24);
          Check_Margin (y2, y1, 25);

          xView := x2 - x1;  xOfs := width  - xView;
          yView := y2 - y1;  yOfs := height - yView;

          Check_Margin (xpos, 1, xOfs);
          Check_Margin (ypos, 1, yOfs);

          position_page(handle, xpos, ypos);
          view_page    (handle, x1, y1, x2, y2);
        end;
      end;
    end;
  end;
until (quit_chg or (opt <> All));
Disappear_view (ChgScr.handle);
end;
```

```
{=============================================================}
{                                                             }
{ PC-TECH JOURNAL,        MARCH 1986,        PROGRAMMING PRACTICES  }
{                                                             }
{ TITLE: TAKING COMMAND IN TURBO PASCAL                 PAGE 161  }
{                                                             }
{=============================================================}
{                                                             }
{           SUB-PROCESS AND MEMORY MANAGEMENT LIBRARY         }
{           -----------------------------------------         }
{                                                             }
{        REQUIRE TWO FILE TO COMPILE THIS MODULE, THEY ARE    }
{                  DOS4AH.COM, DOS4BH.COM                     }
{                                                             }
{           A DEMO PROGRAM IS AVAILABLE, WHICH IS NAMED       }
{                        DOSCOM.PAS                           }
{                                                             }
{=============================================================}
{ ----- not used -------
type r8086 = record
               ax, bx, cx, dx, bp, si, di, ds, es, flags: integer;
             end;
var  regs  : r8086;
  ---------------------- }

type asciiz = string[ 65 ];

{ Current Drive                                               }
{-------------------------------------------------------------}
function    Current_Drive: integer;
  begin
    regs.ax := $1900;
    msdos(regs);
    Current_Drive := lo(regs.ax);
  end;


{ Current Directory                                           }
{-------------------------------------------------------------}
var d_str : asciiz;

function    Current_Dir (drive : integer;
                         Var Dir_str : asciiz): integer;
  var i     : byte ABSOLUTE dir_str;

  begin
    regs.ax := $4700;
    regs.ds := Seg(Dir_str);
    regs.si := Ofs(Dir_str[1]);
    regs.dx := drive;
    msdos(regs);
    if (regs.flags and 1) <> 0 then
        Current_Dir := lo(regs.ax)
    else begin
        { determine & set LENGTH of DIR_STR }
        i := 1;
        while ( (dir_str[i] <> chr(0)) and (i < SizeOf(dir_str)) )
            i := i + 1;
        i := i - 1;

        Current_Dir := 0;
    end;
  end;


{ Prompt with Current Drive & Dir                             }
{-------------------------------------------------------------}
function    dir_prompt (Var Prompt_str : asciiz): integer;
  begin
    dir_prompt := current_dir (0, prompt_str);
```



```
    prompt_str := chr(65 + current_drive) + ':\' + prompt_str;
  end;


{ Maximum MEMORY available                                    }
{-------------------------------------------------------------}
function    Memory_Avail: integer;
    begin
      regs.es := Cseg;
      regs.ax := $4a00;
      regs.bx := $ffff;

      msdos(regs);
      Memory_Avail := regs.bx Div 64;
    end;


{ REDUCE MEMORY ALLOCATION                                    }
{-------------------------------------------------------------}
function    dos4AH(pp_to_release: integer): integer;
    external 'spawn4AH.bin';


{ EXECUTE A SUB PROCESS                                       }
{-------------------------------------------------------------}
function    dos4BH(var program_name, parameter_string): integer;
    external 'spawn4Bh.bin';


{ ALLOCATE A NEW MEMORY BLOCK                                 }
{-------------------------------------------------------------}
function    dos48H(pp_needed: integer; var block_segment: integer): integer;

  begin
    regs.bx := pp_needed;               { # of paragraphs req.   }
    regs.ax := $48 shl 8;               { Function call 48H.     }
    msdos(regs);                        { Call DOS               }
    if (regs.flags and 1) <> 0 then begin{ Is carry flag set     }
        block_segment := regs.bx;       { Yes, return avail. pps }
        dos48H        := lo(regs.ax);   { and error number       }
    end
    else begin
        block_segment := regs.ax;       { No, return seg. address}
        dos48H        := 0;             { and error code 0.      }
        end;
    end;


{ RELEASE A MEMORY BLOCK                                      }
{-------------------------------------------------------------}
function    dos49H(block_segment: integer): integer;

  begin
    regs.es := block_segment;       { segment address to release. }
    regs.ax := $49 shl 8;               { Func. call 49H.        }
    msdos(regs);                        { Call DOS.              }
    if (regs.flags and 1) <> 0 then     { Is carry flag set      }
        dos49H := lo(regs.ax)           { Yes, ret. error no.    }
    else
        dos49H := 0;                    { No, ret. error code 0  }
  end;


{ OBTAIN A PROCESS'S EXIT CODE                                }
{-------------------------------------------------------------}
```

```
function    dos4DH: integer;

  begin
    regs.ax := $4d shl 8;                { Function call 4dH      }
    msdos(regs);                         { Call DOS               }
    dos4DH  := lo(regs.ax);              { Return exit code       }
  end;

{ GET COMMAND PROCESSOR NAME                                       }
{------------------------------------------------------------------}

function    get_comspec(var comspec: ascii1r): boolean;

type
    dos_env_string =^dos_env_type;
    dos_env_type   =array [1..254] of byte;

var
    dos_env:    dos_env_string;
    dos_envs:   string [255];
    idx:        integer;

  begin
    get_comspec   := false;
    dos_env       := ptr(memw[cseg:$2c], $0);       { Get 254 bytes of the  }
    move(dos_env^, dos_envs[1], 254);               { DOS environ. string   }
    dos_envs[255] := #0;
    dos_envs[0]   := #255;
    idx           := pos('COMSPEC=', dos_envs);     { Find COMSPEC= portion }
    if idx= 0 then begin
        writeln('*** "COMSPEC=d:[path]filename" not in DOS.');
        get_comspec := true;
        exit;
    end
    else begin
        delete(dos_envs, 1, idx+7);                 { Isolate ASCIIZ string }
        idx       := pos(#0, dos_envs);             { drive[path]filename   }
        dos_envs := copy(dos_envs, 1, idx);         { of com. processor     }
        while dos_envs[1]=' ' do
            delete(dos_envs, 1, 1);
        comspec  := dos_envs;                       { Return FALSE          }
    end;
  end;

{ HANDLE A DOS ERROR CONDITION                                     }
{------------------------------------------------------------------}

function    dos_error_check(error_code: integer): boolean;

    type
        error_table_type= array[1..18] of string[41];

    const
        error_table: error_table_type=     { RANGE: 1 TO 18 DEC.  }
            ('Invalid function number',
            'File not found',
            'Path not found',
            'Too many open files (no handles left)',
            'Access denied',
            'Invalid file handle',
            'Memory control blocks destroyed',
            'Insufficient memory',
            'Invalid memory block address',
            'Invalid environment',
            'Invalid format',
            'Invalid access code',
            'Invalid data',
            'UNRECOGNIZED ERROR',
```



```
            'Invalid drive was specified',
            'Attempted to remove the current directory',
            'Not same device',
            'No more files');

  begin
    dos_error_check := true;
    if error_code= 0 then
        dos_error_check := false
    else
        writeln('*** DOS error ', error_code, ': ', error_table[error_code]);
  end;
```

```
type  st_type = string[255];

function B_str (B_num : byte; len : byte): st_type;
var   st      : st_type;
begin
  str (B_num : len, st);  B_str := st;
end;


function I_str (I_num : integer; len : byte): st_type;
var   st      : st_type;
begin
  str (I_num : len, st);  I_str := st;
end;


function R_str (R_num : real; menissa, decimal : byte): st_type;
var   st      : st_type;
begin
  str (R_num : menissa : decimal, st);  R_str := st;
end;

{.pa}

function S_str (S_st       : st_type;          { ASCII string  }
                L_or_C_or_R : char;            { Left, Center or Right justify }
                len        : byte;             { length of output string }
                fill_ch    : char):    st_type; { fill character }
var i,       s_len,
    b_start, b_end,
    s_start, s_end   : byte;
    start            : integer;
    st               : st_type;
begin
  s_len := length (S_st);
  start := len - s_len;
  if ( (start < 0) or (s_len >= 80) or (len >= 80) ) then begin
    st[0] := chr(len);
    for i:=1 to len do
      st[i] := Fill_Ch;
  end
  else begin
    if (upcase(L_or_C_or_R) = 'L') then begin
      s_start := 1;              s_end := s_len;
      b_start := s_end + 1;      b_end := len;
    end
    else if (upcase(L_or_C_or_R) = 'C') then begin
      b_start := 1;              b_end := len;
      s_start := 1 + start div 2;  s_end := s_start + s_len - 1;
    end
    else begin
      b_start := 1;              b_end := start;
      s_start := b_end + 1;      s_end := len;;
    end;

    st[0] := chr(len);             { set the length of the output string }
    for i:=b_start to b_end do     { fill the non-used spaces with fill char }
      st[i] := Fill_Ch;
    for i:= s_start to s_end do    { copy the source string to the output str}
      st[i] := S_st[i - s_start + 1];
  end;

  S_str := st;
end;

{.pa}

Var DispV_BG,              { original  background color }
    DispV_FG,              {           foreground color }
    DispV_BG1,             { new msg   background color }
```



```
    DispV_FG1,             {           foreground color }
    DispV_BG2,             { new value background color }
    DispV_FG2 : integer;   {           foreground color }

procedure DispValue (Msg : st80; Vtype : char; Var Value; len, dec: integer);
var Bvalue : byte    absolute Value;
    Ivalue : integer absolute Value;
    Rvalue : real    absolute Value;
    Str    : st80    absolute Value;
begin
  write(' ');
  TextColor      (DispV_FG1);
  TextBackGround(DispV_BG1);
  write(Msg);
  TextColor      (DispV_FG2);
  TextBackGround(DispV_BG2);
  case Vtype of
    'B' : write(Bvalue:len);
    'I' : write(Ivalue:len);
    'R' : write(Rvalue:len:dec);
    'S' : write(Str   :len);
  end;
  TextColor      (DispV_FG);
  TextBackGround(DispV_BG);
end;
```

```
{.N ctaqpib}
{$V-}
{$K-}
{$R-}

Program D_CAD;
{ ------------------------------------------------------------------------ }
{ Written by  : Alex Tsui                                                  }
{ Date        : 10:00:00 11/01/1986                                        }
{ Last Update : 10/05/88                                                   }
{ ------------------------------------------------------------------------ }
{ Title       : CADA-LDA Interactive Data Acquisition System               }
{             : for DISA   Laser Doppler Anemometer,                       }
{             :      DISA   Four Axes Traversing unit and                  }
{             :    . Cyborg ISAAC Analog I/O system                        }
{ ------------------------------------------------------------------------ }

{ ------------------------------------------------------------------------ }
{ Procedure Debugging Aid                                                  }
{ ------------------------------------------------------------------------ }
{$I   ER-STACK.inc }

{ ------------------------------------------------------------------------ }
{ National Instruments IEEE-488 interface                                  }
{ ------------------------------------------------------------------------ }
{.N ctaqpib}

{$I   dd-NI488.drv }                    { include : declactions for GPIB-PCIIA}
{$I   dd-GPIB.drv  }
{ ------------------------------------------------------------------------ }
{ DOS subprocess / LIBRARY call var. declaration                           }
{ ------------------------------------------------------------------------ }
type  regpack  = record case integer of
                   1: (ax,bx,cx,dx,bp,si,di,ds,es,flags: integer);
                   2: (al,ah,bl,bh,cl,ch,dl,dh         : byte);
                 end;
var   regs     : regpack;
var   Bell     : char;

{.pa}

{.N MainModule}

{ ------------------------------------------------------------------------ }
{ Window declaration                                                       }
{ ------------------------------------------------------------------------ }
const num_of_windows = 6;
type  Windows = (User, Input, Device, LDA, Help, Time_window);
type  st20              = string[20];
      Window_Limits_Rec = record
                            N : st20;
                            A : Array[0..15] of Integer;
                          end;

const  Window_Limits     : Array[Windows] of Window_Limits_Rec
         {hle, wi, he, x1, y1, x2, y2, xv, yv, xo, yo, fG, bG, xpos, ypos, frame}
       =({(N: 'User Window ';
          A: {0,  79,25,   2, 14,  79,23, 0, 0, 0, 0, black, green,    1, 16, 0)),
          (N: 'Input Window ';
           A: {0,  79,25,   31,14,  78,18, 0, 0, 0, 0, black, cyan,    1, 18, 0)),
          (N: 'Device Window ';
           A: {0,  79,25,   2,  3,  79,5,  0, 0, 0, 0, white, blue,    1, 1 , 0)),
          (N: 'LDA Window ';
           A: {0,  79,25,   2,  7,  79,11, 0, 0, 0, 0, black, blue,    1, 1 , 0)),
          (N: 'Help Window ';
           A: {0,  79,435,  2,  4,  79,20, 0, 0, 0, 0, white, black,   1, 1 , 0)),
          (N: 'Time & Date Window ';
           A: {0,  80,1,    1,  1,  80,1,  0, 0, 0, 0, white, red,     1, 1 , 0)));

{.pa}
```



```
{ ------------------------------------------------------------------------ }
{ Command Interpreter Program. (Global Variables)                          }
{ ------------------------------------------------------------------------ }

CONST max_no_cmd = 19;
      ValIdCmd_s : array[0..max_no_cmd] of string[3]
                 =('TRA', 'LDA', 'AIN', 'AOU', 'ALL', 'BEL',
                   'DEB', 'DIS', 'DOS', 'DO:', 'ENA', 'END', 'EXE',
                   'EXI', 'FIL', 'HEL', 'REA', 'QUI', 'WAI', 'WIN');

TYPE  ValidCmd_t =(c_trav,  c_lda, c_ainsc, c_aous,   c_all,  c_bell,
                   c_debug, c_disable, c_dos, c_do,    c_enable, c_end,  c_execute,
                   c_exit,  c_file,    c_help,c_read,  c_quit,   c_wait, c_window);

      st3       = string[3];
      st2       = string[2];
      st80      = string[80];

Type   dev_cpt      = (read_d, set_d, clear_d, init_d);
Const  dev_cpt_char : array[0..3] of char
                    = ('R', 'S', 'C', 'I');

var enable : record
               lda     : boolean;
               tra     : boolean;
               AI      : boolean;
               AO      : boolean;
             end;

Var   _help_loaded,
      _help_opened    : boolean;
      cmd_file        : array [0..10] of text;
      cmd_file_exist  : array [0..10] of boolean;
      cmd_num         : integer;
      DOS_mem         : real;

{.pa}

{ ------------------------------------------------------------------------ }
{ System Utilities Routines. Required above type declarations              }
{ ------------------------------------------------------------------------ }
procedure printf (message : st128);
  begin
    write (message);
  end;


procedure println (message : st128);
  begin
    writeln (message);
  end;


{$I   spawn.inc          }    { Include:  routine to create subprocss }
{$I   cli-dos.drv        }

{$I   Library.inc        }
{$I   STRING.inc         }

{$I   ram-page.inc       }    { Include:  Ram Page Window package     }
{$I   RWindow.inc        }    { Include:  Window routine developed    }
                              {           from RamPage                }

const   TitleScr : RamPageRec
                 = ( name  : 'Title';
                     handle: 0;
                     width : 80;    height:  1;
                     x1    : 1;     y1    : 25;    x2  : 80;  y2  : 26;
                     xView : 0;     yView :  0;    xOfs:  0;  yOfs:  0;
```

```
                    foreGround : white;          backGround : red;
                    xPos : 1;      yPos : 1;     frame_byte : 0 );

       Title_text = ' CADA-LDA : Computer Aided Data Acquisition for DISA''s LDA Environment ';

{ ------------------------------------------------------------------------}
{ Window routines                                                         }
{ ------------------------------------------------------------------------}
procedure Open_Dummy_Window;
  begin
    Appear_View (TitleScr.handle);
  end;

{.pa}

{ ---------------------------------------------- }
VAR time,     date,
    old_time, old_date : time_date_string;
    td_ctr             : integer;

procedure update_Time_Date (immedate : boolean);
  var old_Row,  old_Col  : byte;
      mem : real;

   BEGIN
     time := Get_time;
     if (immedate or (time <> old_time)) then begin
       Open_Dummy_Window;
       old_time := time;
       date     := Get_date;
       mem := MemAvail;
       if (mem < 0) then mem := mem + 65536.0;
       mem := mem * 16.0 / 1024.0;
       with W[time_window] do begin
         Ram_page          (handle);
         Disable_view      (handle);
         TextColor         (foreground);
         TextBackGround    (background);
         mem := MemAvail / 64.0;
         GotoXY( 3,1);     printf ('Date : ' + Date);
         GotoXY(22,1);     printf ('DOS Avail: ' + R_str( DOS_mem,5,0) + 'K');
         GotoXY(42,1);     printf ('LDA Avail: ' + R_str( mem,    5,0) + 'K');
         GotoXY(64,1);     printf ('Time : ' + Time);
         Enable_view       (handle);
         Appear_View       (handle);
         td_ctr := 0;
       end;
     end;
     td_ctr := td_ctr + 1;
   END;

{.pa}

Procedure Open_device_window;
   BEGIN
     Open_Dummy_Window;
     with W[device] do begin
       Ram_page   (handle);
       TextColor  (foreground);  TextBackGround (backGround);
     end;
   END;


Procedure Open_LDA_window;
   BEGIN
     Open_Dummy_Window;
     with W[LDA] do begin
       Ram_page      (handle);
       Appear_Window (W[LDA]);
       TextColor     (foreground);  TextBackGround (backGround);
     end;
   END;
```



```
Procedure Open_user_window;
   BEGIN
     Open_Dummy_Window;
     with W[user] do begin
       Ram_page      (handle);
       Appear_Window (W[USER]);
       TextColor     (foreground);  TextBackGround (backGround);
     end;
   END;


Procedure Open_cmd_window;
   BEGIN
     Open_Dummy_Window;
     with W[input] do begin
       Ram_Page      (handle);
       Appear_Window (W[INPUT]);
       TextColor     (black);  TextBackGround (backGround);
     end;
   END;

{.pa}

Procedure Open_Windows;
  BEGIN
     Window (1,1, 80,25);
     TextColor (white);
     clrscr;
     Init_ram_page;
     Init_chg_scr;

     Init_Window (TitleScr,          true,   false);
     printf       (Title_Text);

     Init_Window_Limits;

     Init_Window (W[time_window],    true,   false);
     Init_Window (W[device],         true,   true);
     Init_Window (W[LDA],            true,   true);
     Init_Window (W[user],           true,   true);     GotoXY (1,25);
     Init_Window (W[input],          true,   true);     GotoXY (1,25);

     if (NOT _help_opened) then begin
       Init_Window (W[help], false, true);
       _help_opened := true;
     end;

     Update_Time_Date (true);
     Open_user_window;
  END;


Procedure Display_All_Windows;
  begin
    Appear_Window (W[time_window]);
    Appear_Window (W[device]);
    Appear_Window (W[LDA]);
    Appear_Window (W[user]);
    Appear_Window (W[input]);
  end;

{.pa}

{ ------------------------------------------------------------------------}
{ Command Line Interpreter for the above drivers                          }
{ ------------------------------------------------------------------------}
```

```
{.M MainModule}

Procedure Exit_to_Cli;   Forward;

Procedure AbortCLI (buf : st80);
  var  temp_ptr : Msg_ptr;
  BEGIN
     Push_msg (' ** ' + buf);
     Open_User_Window;

     printfln ('');
     printfln ('## possible couse of error ##');
     temp_ptr := first_msg;
     while (temp_ptr <> nil) do begin
        printfln (temp_ptr^.msg);
        temp_ptr := temp_ptr^.next;
     end;

     printfln ('Type HELP for more information' + bell);
     Exit_to_Cli;
     Open_cmd_window;
     Pop_Msg;
  END; { AbortCLI }

Procedure DisplayError (err_code : byte);
  BEGIN
     CASE err_code OF
        1 : AbortCLI ('invalid data specified');
        2 : AbortCLI ('no data specified');
        3 : AbortCLI ('invalid sub_command');
        4 : begin
              printfln ('Invalid command' + bell);
              printfln ('Type HELP for more info.');
              Exit_to_Cli;
            end;
     end
  END; { DisplayError }

{.pa}

Procedure UpdateWindow;   Forward;

{ ------------------------------------------------------------------------}
{ GPIB communication drivers                                              }
{ ------------------------------------------------------------------------}
{.M ctagpib}

{$I dd-trav.drv  }
{$I dd-lda.drv   }
{$I dd-isaac.drv }

{$I cli-lda.drv  }
{$I cli-tra.drv  }

{.pa}

{.M MainModule}

{ ------------------------------------------------------------------------}
{ Window Display routine the above drivers                                }
{ ------------------------------------------------------------------------}
Procedure UpdateWindow;
  VAR   i      : byte;
        buf,
        on_off: string[10];
        lines : byte;
        bg    : integer;

  BEGIN
```



```
     Open_device_window;
     Disable_view (W[device].handle);
     ClrScr;

     DispV_FG  := W[device].foreground;
     DispV_FG1 := Yellow ;    { White; }
     DispV_FG2 := White  ;

     DispV_BG  := W[device].background;
     DispV_BG1 := DispV_BG ; { Cyan; }
     DispV_BG2 := DispV_BG ; { Cyan; }

     TextColor      ( DispV_FG );
     TextBackGround ( DispV_BG );
     lines := 0;

     if (enable.lda) then begin
        lines := lines + 1;
        printf ( 'LDA (' + B_str(_LDA.Mode,1) + '):' );
        DispValue('Nsets=', 'I', _LDA.nsets ,   9, 0);
        DispValue('Int. =', 'R', _LDA.Interval, 9, 4);
        DispValue('DMA #=', 'B', _LDA.DMA   ,   9, 0);
        DispValue('Tb   =', 'R', _LDA.TimeBase, 9, 7);
        printfln ('');
     end;

     if (enable.trav) then begin
        lines := lines + 1;
        printf ('TRAVER :');
        DispValue('X    =', 'R', _trav[0].posn, 9, 1);
        DispValue('Y    =', 'R', _trav[1].posn, 9, 1);
        DispValue('Z    =', 'R', _trav[2].posn, 9, 1);
        DispValue('R    =', 'R', _trav[3].posn, 9, 1);
        printfln ('');
     end;

     if (enable.AI) then with _AI do begin
        lines := lines + 1;
        printf ('A. In  :');
        DispValue('Low  =', 'I', low  , 9, 1);
        DispValue('High =', 'I', high , 9, 1);
        DispValue('Count=', 'I', count, 9, 1);
        DispValue('Rate =', 'I', rate , 9, 1);
        printfln ('');
     end;

     if (enable.AO) then with _AO do begin
        lines := lines + 1;
        printf ('A. Out :');
        DispValue('Chan.=', 'I', chan , 9, 1);
        DispValue('DAC  =', 'I', dac  , 9, 1);
        printfln ('');
     end;

     Enable_view (W[device].handle);

     { ------------------------------------------------------------ }
     if ((lines - 1) <> W[device].yView) then begin
        if (lines <= 1) then lines := 1
        else                 lines := lines - 1;
        with W[device] do begin
           Y2 := Y1 + lines;
           yView := lines;
           View_page (handle, x1, y1, x2, y2);
           Position_page (handle, xpos, ypos);
        end;
     end;
     { ------------------------------------------------------------ }

     Appear_Window   (W[device]);
```

```
  END; { UpdateWindow }

{.pa}

{ - - - - Clean Input Command Routine  - - - - - - - - - - - - - - - - - - -}
{         Get Input            Routine                                      }

{$I get-in.inc}

{ - - - - Do Loop Interpreter Routine  - - - - - - - - - - - - - - - - - - -}
{         \Wait    Interpreter Routine                                     }
{         |Execute Interpreter Routine                                     }

{$I cli-dwe.drv}

{ - - - - Online Help Routine  - - - - - - - - - - - - - - - - - - - - - - -}

{$I cli-help.drv}

{.pa}

{- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
VAR data_file : array[0..9] of TEXT;
    file_rec  : array[0..9] of RECORD
                                 device    : char;
                                 open      : boolean;
                                 format    : string[80];
                               END;

Procedure cli_STORE_DATA_FILE (file_num : integer;
                               title    : boolean);        { title or data }
  VAR format_str               : st128;
      line                     : string[255];
      form_buf                 : st10;
      date_str, time_str       : string[8];
      i, j, space, chan, v_code : integer;
  BEGIN
   Push_msg (' Proc: cli_STORE_DATA_FILE');
     format_str := file_rec[file_num].format;
     IF (title) then  line := 'T'
     ELSE             line := 'D';

     WHILE ( FoundDATA (format_str, form_buf)) do begin
        CASE (form_buf[1]) OF
          'D' : begin                                                { DATE }
                  date_str := Get_DATE;
                  IF (title) then
                     line := line + S_str('|___Date','R',9,' ')
                  ELSE
                     line := line + S_str(date_str,  'R',9,' ');
                end;

          'T' : begin
                  IF (form_buf[2] = 'I') then begin                  { TIME }
                     time_str := Get_TIME;
                     IF (title) then
                        line := line + ' |___Time'
                     ELSE
                        line := line + S_str (time_str,  'R',9,' ');
                  end
                  ELSE begin                                         { TRAV }
                     IF (title) then
                        line := line + ' |__X_axis'
                                     + ' |__Y_axis'
                                     + ' |__Z_axis'
                                     + ' |__R_axis'
                     ELSE
                        line := line + R_str (_trav[0].posn, 10, 4)
                                     + R_str (_trav[1].posn, 10, 4)
```



```
                                     + R_str (_trav[2].posn, 10, 4)
                                     + R_str (_trav[3].posn, 10, 4);
                  end;
                end;

          'S' : case (form_buf[2]) of
                  'P' : begin                                        { Space }
                          i := POS ('=', form_buf);
                          IF (i = 0) then
                             line := line + ' '
                          ELSE begin
                             VAL (form_buf[i + 1], space, v_code);
                             IF ( (v_code <> 0) or (space < 0) ) then
                                AbortCLI ('invalid space selection')
                             ELSE
                                line := line + S_str ('','R',space,' ');
                          end;
                        end;
                end;

          'L' : begin                                                { LDA  }
                  format_str := '';
                  IF (title) then
                     line := line + LDA_Output (title, 0)
                  ELSE begin
                     for i:=0 to _LDA.SetRead-2 do begin
                       writeln (data_file[file_num],
                                line + LDA_Output (title, i) );
                     end;
                     line := line + LDA_Output (title, i+1);
                  end;
                end;

          'A' : begin                                                { Analog }
                  if (form_buf[2] = 'O') then begin                  { Output }
                    line := line + AOUS_output (title)
                  end
                  else if (form_buf[2] = 'I') then begin             { Input  }
                    format_str := '';
                    IF (title) then
                      line := line + AINSC_output (title, 0)
                    ELSE begin
                      if (_AI.count = 1) then
                        i := -1
                      else begin
                        for i:=0 to _AI.count-2 do begin
                          writeln (data_file[file_num],
                                   line + AINSC_output (title, i) );
                        end;
                      end;
                      line := line + AINSC_output (title, i+1);
                    end;
                  end;
                end;

        end; { case }
      end; { while }

      IF (title) then
        writeln (data_file[file_num], line + '|')
      ELSE
        writeln (data_file[file_num], line );

      IF (file_rec[file_num].device = 'C') then
        writeln;

    Pop_msg;
  END; { cli_STORE_DATA_FILE }

Procedure cli_FILE  (token   : st3;          { Format, Open, Close or Store }
```

```
                    VAR buf : st128);          { sub commands                }
VAR istart, iend, ilen   : integer;
    i_data, v_code       : integer;
    data                 : st10;
    format_str           : st128;
    file_num, i          : integer;
    file_name            : string[30];
    file_n3              : st3;
BEGIN
   Push_msg (' Proc: cli_FILE');
      CASE token[1] OF
      'O' : begin                                                  { Open }
               iend := POS (',', buf);
               IF (iend = 0) then
                  AbortCLI ('File Name not found')
               ELSE begin
                  file_name := COPY (buf, 1, iend -1);

                  DELETE (buf, 1, iend);

                  IF (buf[1] <> 'T') then
                     AbortCLI ('Expecting "TO FILE=<file_num>')
                  ELSE begin
                     iend := POS ('=', buf);
                     DELETE (buf, 1, iend);
                     IF (not FoundData (buf, data)) then
                        AbortCLI ('File_number not found')
                     ELSE begin
                        VAL (data, file_num, v_code);
                        IF (v_code <> 0) then
                           AbortCLI ('invalid File Number')
                        ELSE begin
                           file_rec[file_num].open := TRUE;

                           FOR i := 1 to iend-1 do
                              file_name[i] := UPCASE (file_name[i]);
                           file_n3 := file_name;
                           IF (file_n3 = 'CON') then
                              file_rec[file_num].device := 'C'
                           ELSE IF (file_n3 = 'PRN') then
                              file_rec[file_num].device := 'P'
                           ELSE
                              file_rec[file_num].device := 'F';

                           ASSIGN (data_file[file_num], file_name);
                           REWRITE (data_file[file_num]);
                        end
                     end;
                  end;
               end;
      '?' : begin                                                  { Format }
               istart := POS ('(', buf);
               icnd := POS (')', buf);
               ilen := iend - istart -1;
               IF (ilen > 0) then
                  format_str := COPY (buf, istart+1, ilen);
               DELETE (buf, 1, iend +1);
               IF (buf[1] <> 'T') then
                  AbortCLI ('Expecting "TO FILE=<file_num>" ')
               ELSE begin
                  iend := POS ('=', buf);
                  data := buf[iend+1];
                  VAL (data, i_data, v_code);
                  IF (v_code <> 0) then
                     AbortCLI ('invalid File Number')
                  ELSE begin
                     file_num := i_data;
                     IF (not file_num in [0..9]) then
                        AbortCLI ('File Number too (large or small)');
```



```
                     end;
                  end;
                  IF (file_rec[file_num].open = FALSE) then begin
                     AbortCLI ('File is not currently open');
                  end
                  ELSE begin
                     file_rec[file_num].format := format_str + ';';
                     cli_STORE_DATA_FILE (file_num, TRUE);
                  end;
               end;
      'S' : begin                                                  { Store }
               IF (not FoundData (buf, data)) then
                  AbortCLI ('File_number not found')
               ELSE begin
                  VAL (data, file_num, v_code);
                  IF (v_code <> 0) then
                     AbortCLI ('invalid File Number')
                  ELSE begin
                     IF (file_rec[file_num].open = FALSE) then
                        AbortCLI ('file is not currently open')
                     ELSE
                        cli_STORE_DATA_FILE (file_num, FALSE);
                  end;
               end;
      'C' : begin                                                  { Close }
               IF (not FoundData (buf, data)) then
                  AbortCLI ('File_number not found')
               ELSE begin
                  VAL (data, file_num, v_code);
                  IF (v_code <> 0) then
                     AbortCLI ('invalid File Number')
                  ELSE begin
                     CLOSE (data_file[file_num]);
                     file_rec[file_num].open := FALSE;
                  end;
               end;
      end;
   Pop_msg;
END;

{.pa}

Procedure cli_Read_Write (com    : dev_cpt;     { read_d, set_d or clear_d }
                          device : ValidCmd_t); { device code              }
  VAR   rw_com   : st10;
        read_flg : boolean;
        i        : byte;
  BEGIN
     Push_Msg (' Proc: cli_Read_Write');

        IF (cmd_num >= 0) then
           delay (300);

        CASE device of
          c_all  : begin
                     cli_Read_Write (com, c_lda);
                     cli_Read_Write (com, c_trav);
                     cli_Read_Write (com, c_ainsc);
                   end;
          c_ainsc: if (enable.AI   and (com = read_d)) then begin
                     Read_AINSC;
                   end;
          c_aous : if (enable.AO  and (com = set_d )) then begin
                     Read_AOUS;
                   end;
          c_lda  : if (enable.LDA and (com = read_d)) then begin
                     Read_LDA;
                   end;
```

```
          c_trav : IF (com = read_d) then begin
                     TRAVER (com, 0, _trav[0].posn);
                     TRAVER (com, 1, _trav[1].posn);
                     TRAVER (com, 2, _trav[2].posn);
                     TRAVER (com, 3, _trav[3].posn);
                   end;
        end;
        IF (com in [set_d, clear_d]) and
           (device in [c_all, c_lda, c_trav, c_ainsc, c_aous]) then
              UpdateWindow;
     Pop_Msg;
  END; { cli_Read_Write }

{.pa}

Procedure Exit_to_CLI;
  VAR   i : integer;
  BEGIN
     Dispose_All_Loop;

     FOR i:=0 to cmd_num do begin
        CLOSE (cmd_file[i]);
        cmd_file_exist[i] := FALSE;
     end;
     cmd_num := -1;

     FOR i:=0 to 9 do begin
        IF (file_rec[i].open) then
           CLOSE (data_file[i]);
     end;
  END;

Procedure cli_Enable (device  : ValidCmd_t;
                      on_off  : boolean);
   VAR found : boolean;
   BEGIN
      IF (device = c_all) then  printfln ('');

      IF (on_off) then
         printfln ('Device: "' + ValidCmd_s[integer(device)] + '" enabled')
      ELSE
         printfln ('Device: "' + ValidCmd_s[integer(device)] + '" disabled');

      With enable DO begin
         CASE device of
           c_lda   : lda  := on_off;
           c_trav  : trav := on_off;
           c_ainsc : ai   := on_off;
           c_aous  : ao   := on_off;
           c_bell  : if (on_off) then bell := ^G
                     else             bell := #175;
           c_all   : Begin
                       cli_enable (c_lda,   on_off);
                       cli_enable (c_trav,  on_off);
                       cli_enable (c_ainsc, on_off);
                       cli_enable (c_aous , on_off);
                     end;
         end;
      end;
   END;

{.pa}

Procedure cli_DEBUG (level_str : st3;
                     var buf   : st128);
  VAR i, v : integer;
     data : st10;
  BEGIN
     IF (NOT FoundData (buf, data)) then
        AbortCLI ('incorrect debug level <0..9>')
```



```
     ELSE begin
        VAL (data, i, v);
        debug_488 := i;
        writeln ('Debug level changed to ', debug_488);
     end;
  END; { cli_DEBUG }

function ConvCmd (Var token3: st3; Var CliCmd: ValidCmd_t): boolean;
  var i : byte;
  begin
    i := 0;
    while ( (i <= max_no_cmd) and (ValidCmd_s[i] <> token3)) do i:=i+1;
    if (i <= max_no_cmd) then begin
      ConvCmd := true;
      CliCmd := ValidCmd_t(i);
    end
    else begin
      ConvCmd := false;
    end;
  end;

{.pa}

Procedure cli_TOKEN (token   : ValidCmd_t;
                     Var buf : st128);
  VAR  dev       : ValidCmd_t;
       com       : st10;
       com3      : st3;
       ch        : char;
       cmd_found,
       tmp       : boolean;
  BEGIN
     Push_Msg (' Proc: cli_TOKEN');

       IF (token = c_do) then
          cli_Begin_DO (buf)

       ELSE IF (token = c_end) then
          cli_End_DO (buf)

       ELSE begin
          REPEAT
             IF ( token in [c_read, c_help, c_enable, c_disable] ) then
                cmd_found := FoundData (buf, com)

             ELSE IF (token = c_wait) then begin
                IF (buf[1] = 'K') then
                   cmd_found := FoundData( buf, com)
                ELSE
                   cmd_found := FoundCom( buf, '=', com);
             end
             ELSE
                cmd_found :=  FoundCom (buf, '=', com);

             IF ( cmd_found ) then begin
                com3 := com;
                case token of
                  c_lda    : cli_LDA          (com3,  buf);
                  c_trav   : cli_TRAV         (com3,  buf);
                  c_ainsc  : cli_AINSC        (com3,  buf);
                  c_aous   : cli_AOUS         (com3,  buf);
                  c_read   : begin
                               if (ConvCmd (com3,  dev)) then
                                 cli_READ_WRITE (read_d,dev)
                               else displayError (4);
                             end;
                  c_file   : cli_FILE         (com3,  buf);
                  c_wait   : cli_WAIT         (com3,  buf);
                  c_execute: cli_EXECUTE_1    (com3,  buf);
```

```
                c_enable : begin
                             if (ConvCmd (com3,  dev)) then
                               cli_ENABLE     (dev,  TRUE)
                             else displayError (4);
                           end;
                c_disable: begin
                             if (ConvCmd (com3,  dev)) then
                               cli_ENABLE     (dev,  FALSE)
                             else displayError (4);
                           end;
                c_debug  : cli_DEBUG          (com3,  buf);
              end;
            end;
            IF (KEYPRESSED) then begin
               read (KBD, ch);
               IF (ch = ^[) then begin
                  UpdateWindow;
                  Exit_to_CLI;
               end;
            end;
          UNTIL (not cmd_found);
        end;

        IF (token = c_read) then
           UpdateWindow
        else IF ( token in [c_lda, c_trav, c_ainsc, c_aous] ) then
          cli_Read_Write (set_d, token);

    Pop_Msg;
  END; { cli_TOKEN }

{.pa}

Procedure MAIN;
  VAR  buf            : st128;
       command        : st10;
       com_found      : boolean;
       com3, token3   : st3;
       CliCmd         : ValidCmd_t;
       ch             : char;
       i              : byte;
       dos_param_flag : boolean;
       dispHelp       : boolean;

  BEGIN
    if (ParamCount > 0) then begin
      buf := '';
      dos_param_flag := true;
      for i:=1 to ParamCount do begin
        buf := buf + ParamStr(i) + ' ';
      end;
      clean_input_cmd (buf);
    end

    else begin
      dos_param_flag := false;
    end;

    token3 := '';
    REPEAT
      Open_cmd_window;

      if (dos_param_flag) then begin
        dos_param_flag := false;
        printfln ('');
        printf   ('Init CMD > ' + buf);
      end

      else IF (NOT _do_flg) then begin
        Get_Input ('', buf);
```



```
      token3 := buf;

      IF (token3 = 'DO:') then begin
        if (cli_Store_DO (buf)) then begin
          buf := _do_rec_ptr^.text;
          _do_rec_ptr := _do_rec_ptr^.next;
        end
        else begin
          _do_flg := false;
          buf := ';';
        end;
      end;
    end

    ELSE begin
      buf := _do_rec_ptr^.text;
      _do_rec_ptr := _do_rec_ptr^.next;
      printfln ('');
      write ('Loop:', _do_cur_lab:1, ':',
                      _do[_do_level].ctr:3, ' > ', buf);
    end;
    Update_Time_Date (true);
    Open_user_window;

    token3 := buf;
    dispHelp := false;

    if (buf[1] = ';') then
      begin end                                    { skip line "command"}

    else if ( ConvCmd (token3, CliCmd) ) then begin
      (* A table of valid commands are defined in globle area *)

      if ( CliCmd = c_dos ) then begin
        Ram_page      (0);                         { open system window }
        Appear_view   (0);
        TextColor     (white);
        TextBackground (black);
        Enable_cursor;
        ClrScr;
        delete  (buf, length(buf), 1);
        cli_dos (buf);
        ClrScr;
        Display_All_Windows;
      end

      else if (FoundCom (buf, ':', command)) THEN BEGIN
        IF ( CliCmd in [c_trav,  c_lda,     c_ainsc,  c_aous,
                        c_debug, c_disable, c_do,     c_enable, c_end,
                        c_file,  c_execute, c_read,   c_wait] )
        then begin
          cli_TOKEN (CliCmd, buf);
        end
        else dispHelp := true;
      end

      else IF (CliCmd = c_help) then
        cli_help

      else if (CliCmd = c_window) then begin
        if (NOT _help_opened) then begin
          Init_Window (W[help], false, true);
          _help_opened := true;
        end;

        Change_Screen (User, All);
        Display_All_Windows;
      end

      else if ( CliCmd in [c_exit, c_quit] ) then begin
```

```
          Exit_to_CLI;
        end

        else dispHelp := true;
      end

      else dispHelp := true;

      if (dispHelp) then begin
        open_cmd_window;
        printfln ('');
        printfln ('* Invalid Command'+ bell);
        printf   ('* type HELP for more info.');
      end;

      IF (KEYPRESSED) then begin
        read (KBD, ch);
        IF (ch = ^[) then  Exit_to_CLI;

      end;
    UNTIL (CliCmd = c_quit);
  END; { MAIN }

{.pa}

Procedure Init_Sys_Flgs;
VAR i                 : byte;
    err_msg           : st128;
  BEGIN
    { ---------- define initial global value ---------- }
    Bell          := #175;

    debug_488     := 0;              { no debug message will be disp.}

    _help_loaded  := false;
    _help_opened  := false;

    _do_flg       := FALSE;
    _do[0].first  := nil;

    FOR i:=0 to 9 do
      file_rec[i].open := FALSE;

    cmd_num := -1;
    FOR i:=0 to 10 do
      cmd_file_exist[i] := FALSE;

    { ---------- Initialize System and All Device Drivers ---------- }
    DOS_mem := 0;

    Init_msg_stack;

    SetCrtMode (c80);
    Enable_cursor;
    Open_Windows;

    if ( Init_GPIB ('TRAV', trav_id, err_msg) ) then
      printfln ('cannot init GPIB-TRAV : ' + err_msg);

    Init_trav_drv;       Init_cli_trav;
    Init_ISAAC;
    INIT_CLI_LDA;

    DOS_mem := MemAvail / 64.0;
    Init_dos (memAvail - 4500);                    { paragraph - 64k /16 }
    DOS_mem := DOS_mem - (MemAvail / 64.0);

    Open_CMD_Window;
  END; { Init_Sys_Flgs }
```



```
{.pa}                 ',

var  main_index : byte;

begin
  Init_Sys_Flgs;
  printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
  printfln ('                 Welcome to CADA-LDA             ');
  printfln ('                                                 ');
  printfln ('       Computer Aided Data Acquisition System');
  printfln ('               for the LDA environment           ');
  printfln ('               Written by :  Alex Tsui           ');
  printfln ('       MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
  delay    ( 1500 );

  for main_index:=0 to 15 do printfln ('');

  printfln ('   MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
  printfln ('     Login TIME & DATE > ' + get_time + ' ' + get_date);
  printfln ('   MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');
  printfln ('     Computer Aided Data Acquisition System');
  printfln ('     for :  Laser Doppler Anemometer          ');
  printfln ('            Axes Controller');
  printfln ('            Analog I/O');
  printf   ('   MMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM');

  Main;
  SetCrtMode (c80);

end.
```

CADA-LDA.DEF

```
User Window
        79        25
         2        13        79        23
        77        10         2        15
         0         2
         1        15
       255
Input Window
        79        25
        31        13        78        20
        47         7        32        18
         0         3
         1        18
       255
Device Window
        79        25
         2         3        79         6
        77         3         2        22
        15         1
         1         1
       255
LDA Window
        79        25
         2         8        79        11
        77         3         2        22
        15         1
         1         1
       255
Help Window
        79       520
         2         4        79        20
        77        16         2       504
        15         5
         1         1
       255
Time & Date Window
        80         1
         1         1        80         1
        79         0         1         1
        15         4
         1         1
         0
```


CIL.BAT

```
link /M/L \lc\l\lc+%1 %2 %3,%1,%1/m,\lc\l\lc+\t-lda\isaac\labcl.lib
```

MAKE.BAT

```
tm      start
bigmake cada-l %1 %2 %3 %4
tm      stop
```

APPENDIX E
SAMPLE DATA ACQUISITION PROGRAMS FOR PC-LDA INTERFACE CARD

```
{-------------------------------------------------------------------}
procedure init_lda_interface (count : integer);
var
  command, offset  : array[0..2] of byte;
  i, counter       : integer;
begin
  command[0] := $89;                { 8255-0 pa,pb=output, pc=input }
  command[1] := $9b;                { 8255-1 pa,pb,pc=input         }
  command[2] := $9b;                { 8255-2 pa,pb,pc=input         }
  offset[0] := 0;                   { offset address of 8255-0      }
  offset[1] := 8;                   { offset address of 8255-1      }
  offset[2] := 12;                  { offset address of 8255-2      }

  port[base + 4] := 0;              { issue a software reset        }

  for i:=0 to 2 do                  { load each 8255 mode register  }
    port[base +offset[i] +3] := command[i];
                                    { load count to interface card  }
  if (count >0) and (count <7) then
   begin
    if (count <4) then
     begin
      counter := 6 - count + 1;
      writeln (' count-1 =', counter);
     end
    else
     begin
      counter := 6 - count;
      writeln (' count =', counter);
     end
   end

  else
   begin
    writeln;
    writeln (^G, ' count = ', count:4, '  COUNT ERROR ');
    halt
   end;

  { load and set PC-LDA interface's counter }
  { clear LDA command reg.    dis:DMA dis:INT dis:ARM dis:INH set:count  }


  port[base +offset[0] +0] := $00  or $00  or $00  or $00  or counter;
  port[base +5] := 0;               { set internal counter          }
  port[base +6] := 0;               { clear inhibit and DMA         }

  { enable the ARM and INH flags }
  { load LDA command reg.    dis:DMA  dis:INT en:ARM  en:INH  set:count  }


  port[base +offset[0] +0] := $00  or $00  or $20  or $10  or counter;
end;
```

PC-LDA Initialization Routine

Software Polling Demonstration Program

```
program test;

{ ========== Software Polling ========================================

   This program is used to demonstrate how easily one can read a number of
  data from the LDA-Counter Processor.  The user is required to key in the
  number of the data port to be read (example, 1-> read data port 6 to 6;
  2-> read data port 5 to 6, and so on to 6-> read data port 1 to 6), and
  the number of sets of data.  When all the data is read from the LDA-Counter
  Processor unit through the PC-LDA interface, the entire list of the input
  data will be displayed.

   Now, lets look at the program more closely.  A number of procedures have
  been written here, each of which have a particular function.  The following
  is a table used to explain the procedures.

  PROCEDURE              PURPOSE
  ---------              -------
  1) get_inputs          Read in two values NPORT and NSET.
                         NPORT = # of port one wants to read.
                         NSET  = # of sets of data to be read from the LDA unit.

  2) init_lda_interface  1) Reset all the 8255A-5's.
                         2) Load the 8255's command register with the proper
                            mode and direction code.
                         3) Determine the PC-LDA's counter code from the count
                            input.
                         4) Disable all the interface hardware flags and enter
                            the counter code.
                         5) Reset the interface counter & control logic with
                            new counter value.
                         6) Send a clear INHibit signal to make sure the
                            inhibit (-INH) signal to the LDA-Counter Processor
                            is clear.

  3) disp_title          This routine displays the heading for the data set.

  4) display             This routine is used to display the input data to
                         the screen.

  5) polling_routine     This routine does the actual data acquisition.
                         1) Test the PC-LDA system register number 2 for the
                            status of the DATA READY (+DR) line.
                         2) If it contains a logic one ('1') in bit 7 then the
                            routine will read and store the newly acquired data.
                         3) When all the data are read, a display data routine
                            will display all the input data.

  6) The main program    This uses the previously discussed procedure to
                         1) get input NPORT & NSET from the user.
                         2) Clear the set of old data.
                         3) Initialize to PC-LDA interface.
                         4) Clear the polling counter to zero.
                         5) Then run the polling routine.
}
```



```
const
  test_prog    = false;
  base         = $8300;

type
  st10         \ = string[10];

var
  data                 : array [0..599] of byte;
  i,
  poll_counter,
  nport, nset          : integer;
  command              : char;
  hexcode              : st10;

{------------------------}
{ DEFINITION OF PROCEDURES }
{------------------------}

procedure get_inputs;

begin
  clrscr;
  if (test_prog) then
  begin
    write (^G, ' this is just a test run,');
  end;
  writeln;
  write (' # of port you required to read ? ');
  read (nport);
  writeln; writeln;
  write (' # of set of ports are required ? ');
  read (nset);
  writeln;
end;


{-------------------------------------------------------------------}
procedure init_lda_interface (count : integer);
var
  command, offset  : array[0..2] of byte;
  i, counter       : integer;
begin
  command[0] := $89;                { 8255-0 pa,pb=output, pc=input }
  command[1] := $9b;                { 8255-1 pa,pb,pc=input         }
  command[2] := $9b;                { 8255-2 pa,pb,pc=input         }
  offset[0] := 0;                   { offset address of 8255-0      }
  offset[1] := 8;                   { offset address of 8255-1      }
  offset[2] := 12;                  { offset address of 8255-2      }

  port[base + 4] := 0;              { issue a software reset        }

  for i:=0 to 2 do                  { load each 8255 mode register  }
    port[base +offset[i] +3] := command[i];
                                    { load count to interface card  }
  if (count >0) and (count <7) then
   begin
    if (count <4) then
     begin
      counter := 6 - count + 1;
      writeln (' count-1 =', counter);
     end
    else
     begin
      counter := 6 - count;
      writeln (' count =', counter);
     end
   end
```

```
  else
   begin
    writeln;
    writeln ('G, ' count = ', count:4, '  COUNT ERROR ');
    halt
   end;

  { load and set PC-LDA interface's counter }
  { clear LDA command reg.    dis:DMA dis:INT dis:ARM dis:INH set:count  }

  port[base +offset[0] +0] := $00  or $00  or $00  or $00  or counter;
  port[base +5] := 0;              { set internal counter          }
  port[base +6] := 0;              { clear inhibit and DMA         }

  { enable the ARM and INH flags }
  { load LDA command reg.    dis:DMA  dis:INT en:ARM  en:INH  set:count  }

  port[base +offset[0] +0] := $00  or $00  or $20  or $10  or counter;
end;


{-------------------}
procedure disp_title;

var
  i : integer;

begin
  writeln;
  write (' set#  ');
  for i:=1 to nport do write (' port[',i:2, ']   ');
  writeln;
  write (' <===>');
  for i:=1 to nport do write (' ==========');
  writeln;
end;


{-------------------------------------------------}
PROCEDURE HEX( DATA : INTEGER ;VAR HEXCODE : STR10 );
  CONST
    HEX_STR      : ARRAY [0..15] OF CHAR = ('0','1','2','3','4','5','6','7',
                                            '8','9','A','B','C','D','E','F');
  BEGIN
    HEXCODE := '$' + HEX_STR[ (DATA AND $F0) SHR 4 ]
                   + HEX_STR[ DATA AND $0F ];
  END;


{----------------}
procedure display;

var
  i, j, count  : integer;

begin
  { *** display titles }
  clrscr;
  disp_title;

  { *** initialize data counter }
  count := 0;

  { *** display data }
  for i:=1 to nset do
  begin
    write (' <', i:3, '>');
    for j:=1 to nport do
    begin
      hex (data[ count], hexcode);
```



```
      write ( data[ count]:6, ' =', hexcode:4);
      count := count +1;
      if (count > 599) then
      begin
        writeln; writeln ('G, ' the data array has been exceeded !!');
        exit;
      end;
    end;
    writeln;
    if (frac(i/20) = 0) then
    begin
      writeln;
      write (' press key for more ! ');
      repeat until (keypressed);
      writeln; writeln;
      disp_title;
    end;
  end;
end;


{-----------------------}
procedure polling_routine;

var
  i  : integer;

begin

  repeat
    if (port[base +2] and $80) = $80 then
    begin
      writeln (' Reading data number ', poll_counter:3);
      for i:=1 to nport do
      begin
        data[poll_counter] := port[base +7];
        poll_counter := poll_counter +1;
      end;
      port[base +6] := 0;              { clear INH line }
      port[base +5] := 0;              { send the next ARM pulse }
    end
  until ((poll_counter >= nport*nset) or (keypressed));

  port[base +0] := 0;                  { release lda }

  repeat
    display;
    writeln;    writeln (' Type "y" to display the data one more time,');
                writeln (' or CR to rerun program or "q" to quit ?');
                read (command);
  until (upcase(command) <> 'Y');
end;

{====================}
{ main porgram begin }
{====================}
begin
  repeat
    get_inputs;

    for i:=0 to 599 do                 { clear data value }
      data[0] := 0;

    init_lda_interface(nport);

    poll_counter := 0;
    polling_routine;

    port[ base +1] := 0;                 { release the LDA }
  until (upcase(command) = 'Q');
```

```
end.
```

Demonstration using Interrupt Method

```
{$A+}
program lda_interrupt;

{ ---------------- INTERRUPT METHOD OF DATA ACQUISITION ------------------
```

With the understanding of the software polling program, we now go to a bit more complex and more efficient ways in computing power. The interrupt method of data acquisition technique is employed here. With this, the main program does not spend all of its time waiting for the LDA unit to send in its data, but, a hardware called the interrupt controller (Intel 8259A IC) handles the actual waiting. Another word for waiting is called the hand-shaking, the handshaking signal used in PC interrupt controller is the interrupt request signal (IRQx). When such a signal arrives at the interrupt controller, it in turn informs the microprocessor in the PC system to stop its next executable statement, and jump to an interrupt service routine to service that request. When the service routine is finished, the processor automatically returns to the task it was originally executing.

With this, the user is allowed to do a few things almost simultaneously. However, this method introduces some software overhead for the programmer. One such overhead is to write a service routine which can handle interrupt, and the other is to update the interrupt table. To update the interrupt table, the starting address of the interrupt service routine must be added. This newly added address acts as a jump table to the service routine.

There are a few important points to watch out for when Turbo Pascal is used. Turbo Pascal does not allow any standard I/O procedures or functions to be executed within an interrupt routine because of reentrancy problems, also global variables are not passed. Therefore, read or write statements are not allowed, and all variables passing must be done in absolute memory location. This is shown in the example program.

If one is not familiar with the IBM-PC interrupt system and Turbo Pascal interrupt procedure setup, then special attention must be put to understand the example procedure, namely, "interrupt_routine" & "load_interrupt_pointer", while the rest of the procedure is very much like those found in the soft-ware polling program.

Now, lets look at the "interrupt_routine" procedure. "Base" is defined as a constant, and is equal to the base address of the PC-LDA interface unit. The variable declaration is listed after the constant declaration. This list contains a number of variables which end with "absolute $ffff:ffff". This is the way how Turbo Pascal declares a variable at absolute address $ffff:ffff.

After the variable declaration, one is required to enter the actual interrupt service routine. This is indicated by the "Begin" instruction. When the user is writing an interrupt procedure, he/she must include an inline procedure before any executable statement. This inline procedure is used to save the microprocessor's internal registers. The user is also required to end this routine with another inline procedure, which restores the registers, and also an interrupt return instruction to go back to the main routine.

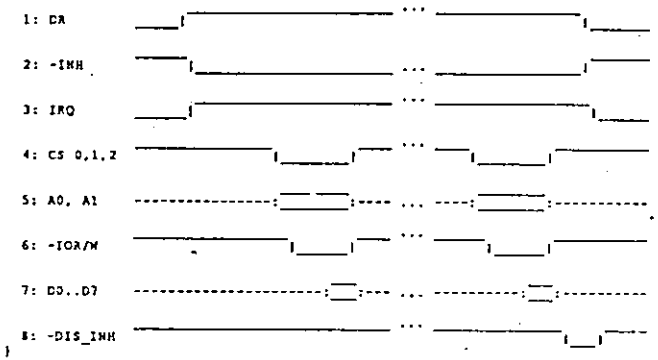
In the "load_interrupt_pointer" procedure, the system interrupt call $21 is used. With this system call one can specify the interrupt type, service routine's segment and offset address and also the load interrupt table function number ($25). Then the user can execute the software interrupt function provided by Turbo Pascal to do the actual table updating.

For the main program the user can execute other procedures without con-tinuously waiting for inputs. In the example given below, a procedure called "wait_and_display" is used to demonstrate the concurrency of interrupt. This procedure simply writes the intr_counter value to the screen, then checks to see if all the data are acquired. If so, it will exit the loop and display the inputs.

The following is the hardware handshaking timing diagram used. It is followed by an example of the program which employes interrupting.

Hardware Information

Sequence of occurrence




```
}
```

```
{-----------------------------------------}
{ GLOBAL DEFINITION OF VARIABLES AND CONSTANT }
{-----------------------------------------}

const
  test_prog    = false;
  base         = $8300;
  intr_channel = 3;

type
  str10        = string[10];

var
  data                : array [0..599] of byte;
  i,
  poll_counter,
  nport, nset         : integer;
  command             : char;
  hexcode             : str10;
  intr_prog_flag : boolean absolute $3000:$0000;
  data_seg       : integer absolute $3000:$0004;
  data_ofs       : integer absolute $3000:$0008;
  intr_counter   : integer absolute $3000:$000c;
  intr_finish    : boolean absolute $3000:$0010;
  max_count      : integer absolute $3000:$0014;
  max_nport      : integer absolute $3000:$0018;

{-----------------------}
{ DEFINITION OF PROCEDURES }
{-----------------------}

procedure get_inputs;

begin
  clrscr;
  if (test_prog) then
  begin
    write ('G, ' this is just a test run.');
  end;
  writeln;
  write (' # of port you required to read ? ');
  read (nport);
  writeln; writeln;
  write (' # of set of ports are required ? ');
  read (nset);
  writeln;
end;


{-------------------------}
procedure interrupt_routine;

const
  base = $8300;

var
  intr_prog_flag : boolean absolute $3000:$0000;
  data_seg       : integer absolute $3000:$0004;
  data_ofs       : integer absolute $3000:$0008;
  intr_counter   : integer absolute $3000:$000c;
  intr_finish    : boolean absolute $3000:$0010;
  max_count      : integer absolute $3000:$0014;
  max_nport      : integer absolute $3000:$0018;
  i  : integer;

begin
  inline ($50/$53/$51/$52/$57/$56/$06/$fb); { save registers & set intr. }

  if (intr_prog_flag) then
```



```
  begin
    if (intr_counter < max_count) then
    begin
      for i:=1 to max_nport do
      begin
        mem[data_seg:data_ofs + intr_counter]. := port[base +7];
        intr_counter := intr_counter +1;
        if (intr_counter >= max_count) then intr_finish := true;
      end;
    end
    else
      intr_finish := true;
  end;

  port[base +6] := 0;                    { clear INH line }
  port[base +5] := 0;                    { send the next ARM pulse }

  port[$20] := $20;                      { send End Of Interrupt to 8259 } {***}

  inline ($07/$5e/$5f/$5a/$59/$5b/$58); { restore registers }
  inline ($8b/$e5/$5d);                  { restore SP & BP }
  inline ($cf)                           { interrupt return }
end;


{-------------------------------}
procedure load_interrupt_pointer;

type
  regpack = record
              ax,bx,cx,dx,bp,di,si,ds,es,flags : integer;
            end;

var
  regpack         : regpack;
  ah, al, temp    : byte;

begin
 { enable the interrupt controller }
 temp := port[$21];                                              {***}
 port[$21] := temp and ($ff xor (1 shl intr_channel));           {***}

 { enter address to interrupt control table via-BIOS func call $25 }
 with regpack do
 begin
  dx := ofs (interrupt_routine);         { load routine offset address }
  ds := Cseg;                            { load routine segment address }
  ah := $25;                             { select "set vector intr. func. call }
  if (intr_channel = 2) then             { load al with interrupt type # }
    al := 10                               { al := 10 if IRQ2 is used }
  else
  begin
    if (intr_channel = 3) then
      al := 11                             { al := 11 if IRQ3 is used }
    else
    begin
      if (intr_channel = 5) then
        al := 13                           { al := 13 if IRQ5 is used }
      else                                 { illegal interrupt are selected }
      begin
        writeln ( 'incorrect interrupt channel, program terminated' );
        halt;
      end;
    end;
  end;

  ax := ah shl 8 + al;                   { redirect hardware interrupt call }
  intr ($21, regpack);                   { make system interrupt call to O.S. }
 end;
end;
```

```
{-------------------------------------------------------------------}
procedure init_lda_interface_and_enable_interrupt(count : integer);

var
  command, offset  : array[0..2] of byte;
  i, counter       : integer;

begin
  command[0] := $89;              { 8255-0 pa,pb=output, pc=input }
  command[1] := $9b;              { 8255-1 pa,pb,pc=input         }
  command[2] := $9b;              { 8255-2 pa,pb,pc=input         }
  offset[0] := 0;                 { offset address of 8255-0      }
  offset[1] := 8;                 { offset address of 8255-1      }
  offset[2] := 12;                { offset address of 8255-2      }

  port[base + 4] := 0;            { issue a software reset        }

  for i:=0 to 2 do                { load each 8255 mode register  }
    port[base +offset[i] +3] := command[i];
                                  { load count to interface card  }
  if (count >0) and (count <7) then
   begin
    if (count <4) then
     begin
      counter := 6 - count + 1;
      writeln (' count-1 =', counter);
     end
    else
     begin
      counter := 6 - count;
      writeln (' count =', counter);
     end
   end
  else
   begin
    writeln;   writeln ('G, ' count = ', count:4, '  COUNT ERROR ');
    halt
   end;

  { clear LDA command reg.    dis:DMA dis:INT dis:ARM dis:INH set:count  }
  port[base +offset[0] +0] := $00  or $00  or $00  or $00  or counter;

  port[base +5] := 0;             { set internal counter          }
  port[base +6] := 0;             { clear interrupt and DMA       }

  { load LDA command reg.    dis:DMA  en:INT  en:ARM  en:INH  set:count  }
  port[base +offset[0] +0] := $00  or $40  or $20  or $10  or counter;
end;


{-------------------}
procedure disp_title;

var
  i : integer;

begin
  writeln;
  write (' set#  ');
  for i:=1 to nport do write (' port[',i:2, ']   ');
  writeln;
  write (' <===>');
  for i:=1 to nport do write (' =========');
  writeln;
end;


{-------------------------------------------------}
```



```
PROCEDURE HEX( DATA : INTEGER ;VAR HEXCODE : ST10 );
  CONST
    HEX_STR      : ARRAY [0..15] OF CHAR = ('0','1','2','3','4','5','6','7',
                                            '8','9','A','B','C','D','E','F');
  BEGIN
    HEXCODE := '$' + HEX_STR[ (DATA AND $F0) SHR 4 ]
                   + HEX_STR[ DATA AND $0F ];
  END;


{----------------}
procedure display;

var
  i, j, count  : integer;

begin
  { *** display titles }
  clrscr;
  disp_title;

  { *** initialize data counter }
  count := 0;

  { *** display data }
  for i:=1 to nset do
  begin
    write (' <', i:3, '>');
    for j:=1 to nport do
    begin
      hex (data[ count], hexcode);
      write ( data[ count]:6, ' =', hexcode:4);
      count := count +1;
      if (count > 599) then
      begin
        writeln; writeln ('G, ' the data array has been exceeded !!');
        exit;
      end;
    end;
    writeln;
    if (frac(i/20) = 0) then
    begin
      writeln;
      write (' press key for more ! ');
      repeat until (keypressed);
      writeln; writeln;
      disp_title;
    end;
  end;
end;


{-------------------------}
procedure wait_and_display;

var
  i, j              : integer;
  status,
  tc, interrupted   : boolean;

begin
  { read interrupt status flag to determine end of interrupt cycle }
  interrupted := false;
  if (not test_prog) then
  begin
    repeat
      writeln ('intr_counter = ', intr_counter:3);              {test}
      status := intr_finish;
      if (keypressed) then
      begin
```

```
        writeln; writeln; writeln ('G, ' PROGRAM HALTED BY USER ');
        interrupted := true;
      end;
    until ((status) or (interrupted));
 end;

 repeat
   port [base + 3] := 0;                { disable the interrupt and free the }
                                        { LDA counter processor              }
   display;
   writeln; write ('Type "y" to display the data one more time, CR to rerun or "q" to quit ?');
            read (command);
 until (upcase(command) <> 'Y');
end;


{====================}
{ main porgram begins }
{====================}
begin
  repeat
    get_inputs;

    max_count := nset*nport;
    max_nport := nport;
    for i:=0 to 599 do                  { clear data value }
      data[i] := 0;

    data_seg := seg( data[0] );
    data_ofs := ofs( data[0] );

    load_interrupt_pointer;

    init_lda_interface_and_enable_interrupt(nport);
    port[ base + 1] := 3;               { set TIME BASE for the interval board }
                                        { 3 --> 100ms                          }
    intr_prog_flag := true;
    intr_finish := false;
    intr_counter := 0;

    wait_and_display;

    intr_prog_flag := false;

  until (upcase(command) = 'Q');
end.
```

Demonstration of the use of the Direct Memory Access Method

```
program lda_dma;

{ ================ Direct-Memory Access Method of Data Acquisition ==========

  Lastly, the most efficient method of data acquisition is Direct-Memory
Access (DMA).  This will give the highest data transfer rate.  With this
method the user must use the sequential read/write port to do the reading/
writing of data.  This method is very similar to the interrupt method, but
it uses a different controller, called the DMA controller (in the PC system
the Intel 8237A-5).  It also requires a special procedu e to program the
controller to acknowledge the DMA request (DREQx) signal.  In this example,
this procedure is named "DMA_set_and_go".

  First, the DMA command register must be set to accept such a DMA request.
The mode which the PC-LDA interface is designed for is the SINGLE BYTE
TRANSFER.  With this, one can achieve a maximum of 500 KByte/sec. data
transfer rate.

  The next thing to do is to load the data buffer address register and the
word count register within the DMA controller with the proper value.  When
all of these are completed, one can unmask that particular DMA channel to
read the desired number of inputs from the LDA-Counter Processor unit.

  The following is an outline of what is required to write such a program
that utilizes the DMA facility, and following that is the program listing.

   1) Request input size.
      ==================
      - NPORT = # of I/O port one wants to read from the interface card.
      - NSET  = # of set of data required to read.
      ** note : Total # of reading is NPORT * NSET.

   2) Initialize LDA-interface card.
      ============================
      - output a software reset pulse @ base+4.
      - initialize each 8255 control register @ base+0+3, +8+3, +12+3.
      - set enable signal and load counter register. (Valid input: 1..6)
      - initialize internal counter.

   3) Initialize DMA registers.
      ========================
      - load DMA mode register with                 ****   defaults   ****
        i  ) mode of transfer.                     {{ single byte mode }}
        ii ) address inc/dec mode.                 {{ inc              }}
        iii) enable/disable auto initialize mode.{{ disable          }}
        iv ) read/write mode.                      {{ write to memory  }}
        v  ) channel #                             {{ channel 1 or 3   }}
      - load data transfer address.
        i  ) page register ( bit 16..19 )
        ii ) buffer starting address ( bit 0..15 )
      - load word count register.
      - unmask the DMA channel.

   4) Wait and display results.
      ========================
      - loop till terminal count is true.
      - display data.
}
```



```
{ DEFINITION OF GLOBAL CONSTANTS AND VARIABLES }
{--------------------------------------------}
const
  test_prog = false;
  base      = $0300;
  dma       = $0;
  dma_page  = $80;
  dma_chan  = 1;

type
  st10      = string[10];

var
  data                : array [0..599] of byte;
  data_seg, data_ofs,
  nport, nset, i,
  cur_word_count      : integer;
  command             : char;
  hexcode             : st10;


{ DEFINITION OF PROCEDURES }
{--------------------------}
procedure get_inputs;
begin
  clrscr;
  if (test_prog) then
  begin
    write (' this is just a test run.');
    writeln (' this will not trigger or wait for any DMA signals');
  end;
  writeln;
  write (' # of port you require to read ? ');
  read (nport);
  writeln; writeln;
  write (' # of set of ports required ? ');
  read (nset);
  writeln;
end;


{-----------------------------------------}
procedure init_lda_interface (count : integer);
var
  command, offset  : array[0..2] of byte;
  i, counter       : integer;

begin
  command[0] := $89;                { 8255-0 pa,pb=output, pc=input }
  command[1] := $9b;                { 8255-1 pa,pb,pc=input         }
  command[2] := $9b;                { 8255-2 pa,pb,pc=input         }
  offset[0] := 0;                   { offset address of 8255-0      }
  offset[1] := 8;                   { offset address of 8255-1      }
  offset[2] := 12;                  { offset address of 8255-2      }

  port[base + 4] := 0;              { issue a software reset        }

  for i:=0 to 2 do                  { load each 8255 mode register  }
    port[base +offset[i] +3] := command[i];
                                    { load count to interface card  }
  if (count >0) and (count <7) then
   begin
    if (count <4) then
     begin
      counter := 6 - count + 1;
      writeln (' count-1 =', counter);
     end
    else
     begin
```

```
      counter := 6 - count;
      writeln (' count =', counter);
     end
   end
  else
   begin
    writeln;
    writeln (^G, ' count = ',count:4, '  COUNT ERROR ');
    halt
   end;

  { load LDA command reg.    en:DMA  en:ARM  en:INH  set:count    }
  port[base +offset[0] +0] := $80 or $20 or $10 or counter;

  port[base +5] := 0;              { trigger the ARM pulse          }
  port[base +6] := 0;              { clear interrupt and DMA        }

end;


{-----------------------}
procedure dma_set_and_go;
var
  dma_command, temp  : integer;

begin
  dma_command := $41 or dma_chan; { =0100 01**B
                                    | || | **- channel #
                                    | || **--- write (to memory) transfer
                                    | |*------ disable autoinitialization
                                    | *------- increment address mode
                                    **-------- single byte transfer       }

  port[dma +11] := dma_command;

  port[dma +12] := dma_command;    { reset first/last flip flop           }

  data_seg := seg( data);          { find segment address of data         }
  data_ofs := ofs( data);          { find offset address of data          }
  writeln (' data segment & offset address= ' ,data_seg, ',  ', data_ofs);

  { *** load page_reg with 1st 4 bits of 20 bit addr }
  writeln; writeln (' DMA channel ', dma_chan:2, ' is selected');
  if (dma_chan = 1) then
  begin
    port[dma_page +3] := (data_seg and $f000) shr 12
  end
  else
  begin
    if (dma_chan = 3) then
      port[dma_page +2] := (data_seg and $f000) shr 12
    else
      writeln (' channel= ', dma_chan, ' SELECTION ERROR ');
  end;
  writeln (' dma_page = ', (data_seg and $f000) shr 12);

  { *** load dma controller with buffer start addr }
  temp := ((data_seg and $0fff) shl 4) + data_ofs;
  port[dma +(dma_chan *2) +0] := lo(temp);
  port[dma +(dma_chan *2) +0] := hi(temp);
  writeln (' buffer start address= ', ((data_seg and $0fff) shl 4) + data_ofs);
  writeln (' lo= ', lo(temp), '   hi= ', hi(temp));

  temp := nport * nset;            { load word count to dma controller    ;
  port[dma +(dma_chan *2) +1] := lo(temp);
  port[dma +(dma_chan *2) +1] := hi(temp);
  writeln (' # of reading= ', temp);
  writeln (' lo= ', lo(temp), '  hi= ', hi(temp));

  if (not test_prog) then
  begin
```



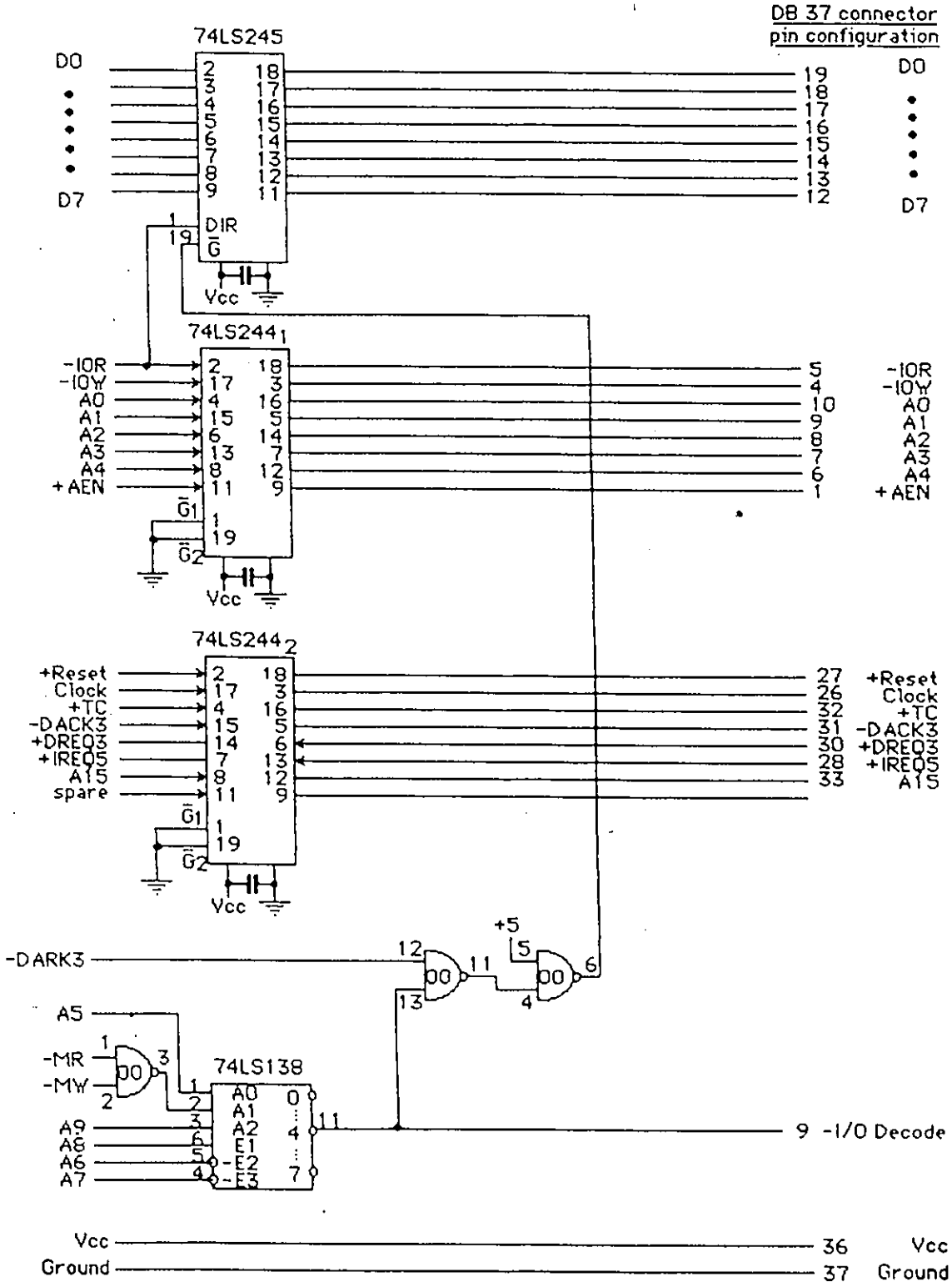
```
    writeln (' unmask channel ', dma_chan, ' of dma controller');
    port[dma +10] := dma_chan;          { write single mask reg to run dma    }
  end;
end;


{-------------------}
procedure disp_title;
var
  i : integer;

begin
  writeln;
  write (' set#  ');
  for i:=1 to nport do write (' port[',i:2, ']   ');
  writeln;
  write (' <---->');
  for i:=1 to nport do write (' ----------');
  writeln;
end;


{-------------------------------------------------}
PROCEDURE HEX( DATA : INTEGER ;VAR HEXCODE : ST10 );
  CONST
    HEX_STR      : ARRAY [0..15] OF CHAR = ('0','1','2','3','4','5','6','7',
                                            '8','9','A','B','C','D','E','F');
  BEGIN
    HEXCODE := '$' + HEX_STR[ (DATA AND $F0) SHR 4 ]
                   + HEX_STR[ DATA AND $0F ];
  END;


{----------------}
procedure display;
var
  i, j, count  : integer;

begin
  { *** display titles }
  clrscr;
  disp_title;

  { *** initialize data counter }
  count := 0;

  { *** display data }
  for i:=1 to nset do
  begin
    write (' <', i:3, '>');
    for j:=1 to nport do
    begin
      hex (data[ count], hexcode);
      write ( data[ count]:6, ' =', hexcode:4);
      count := count +1;
      if (count > 599) then
      begin
        writeln; writeln (^G, ' the data array has been exceeded !!');
        exit;
      end;
    end;
    writeln;
    if (frac(i/20) = 0) then
    begin
      writeln;
      write (' press key for more ! ');
      repeat until (keypressed);
      writeln; writeln;
      disp_title;
    end;
```

```
  end;
end;

{-------------------------}
procedure wait_and_display;
const
  tc_3 = 8;                       { bit patten of TC on channel 3         }
  tc_1 = 2;                       { bit patten of TC on channel 1         }

var
  dma_status,
  i, j              : integer;
  tc, interrupted   : boolean;

begin
  { read dma controller's status register to determine end of dma for ch. 3 }
  interrupted := false;
  if (not test_prog) then
  begin
    repeat
      if (keypressed) then
      begin
        writeln; writeln; writeln ('G, ' PROGRAM HALTED BY USER ');
        interrupted := true;
      end;
      cur_word_count := port[dma +( dma_chan *2) +1] + 1;
      cur_word_count := cur_word_count + port[dma +( dma_chan *2) +1] shl 8;
      writeln ('The current word count is ', cur_word_count:6);
      dma_status := port[dma +8] and $0f;
    until (((dma_status = tc_3) or (interrupted)) or (cur_word_count = 0));
 end;

 repeat
   port[base + 0] := 0;                { release lda -INH signal }
   display;
   writeln; write ('Type "y" to display the data one more time, CR to rerun or "q" to quit ?');
            read (command);
 until (upcase(command) <> 'Y');

end;

{--------------------}
{ main porgram begins }
{--------------------}
begin
  repeat
    get_inputs;
    for i:= 0 to 599 do
      data[i] := 0;
    init_lda_interface(nport);
    port[base +1] :=2;                { set time base to 1 millisec.  }
    dma_set_and_go;
    wait_and_display;
  until (upcase(command) = 'Q');
end.

^Z
```

APPENDIX F - CIRCUIT DIAGRAM OF THE PC-LDA INTERFACE

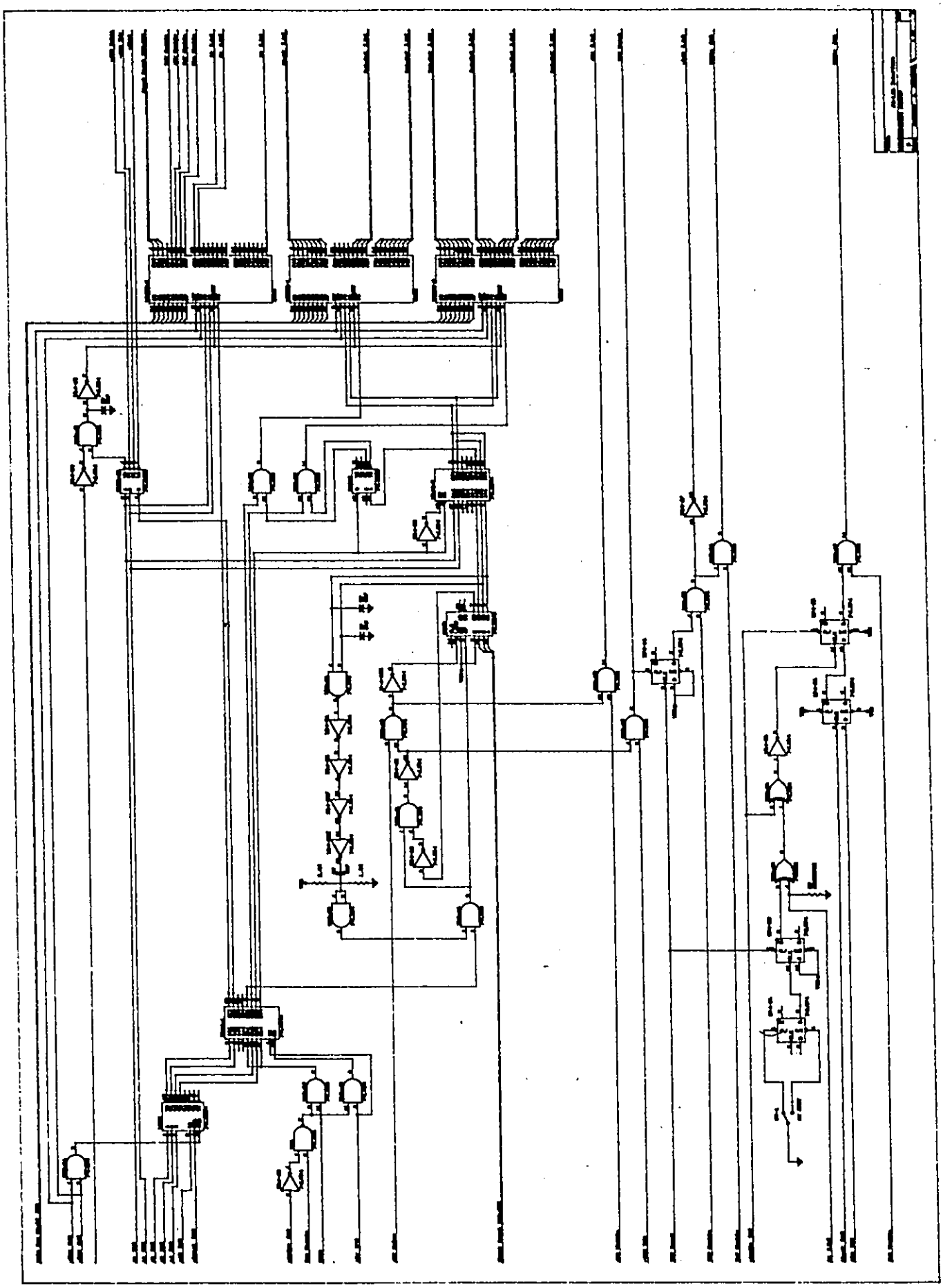
I/O Decode for prototype card

	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
(HEX) 300	1	1	0	0	0	0	0	0	0	0
(HEX) 31F	1	1	0	0	0	1	1	1	1	1
DECODING RANGE	1	1	0	0	0	X	X	X	X	X



Signal Buffer Interface

PC-LDA Circuit Diagram



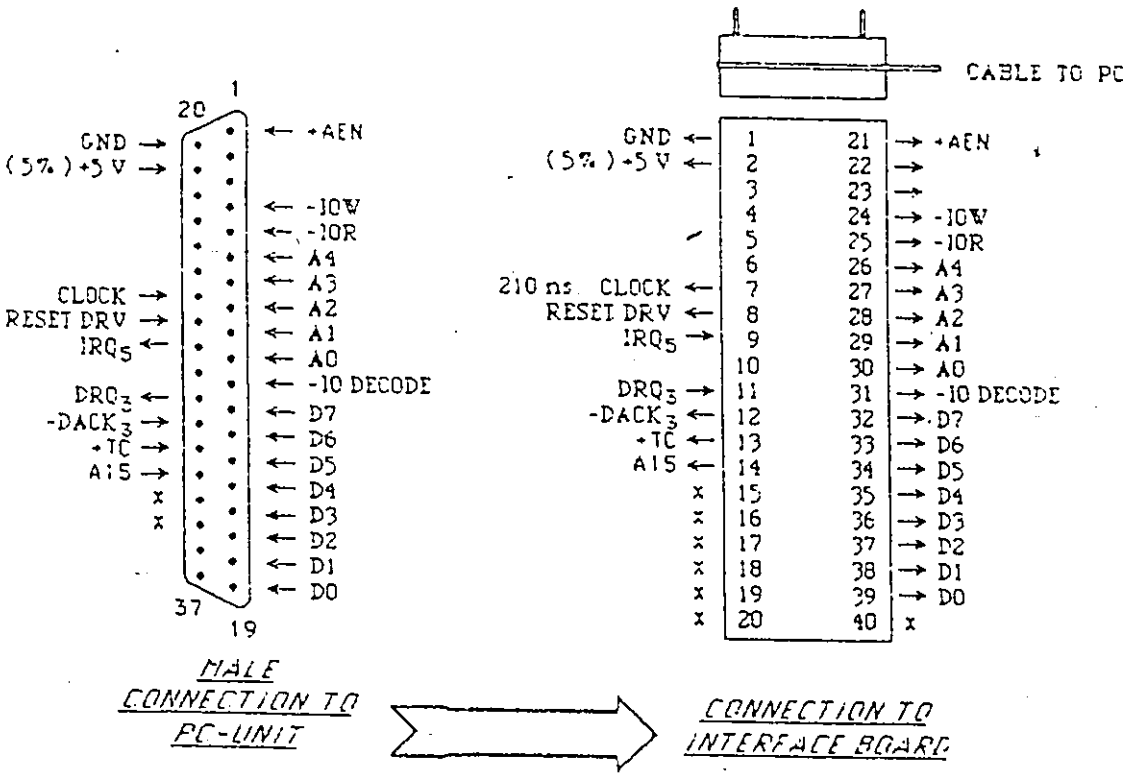
Partlist V1.20 19-SEP-86

Scanning "pclda.dwg"
Scanning Parts "pclda.dwg"
PC-LDA Interface Revised: October 12, 1988

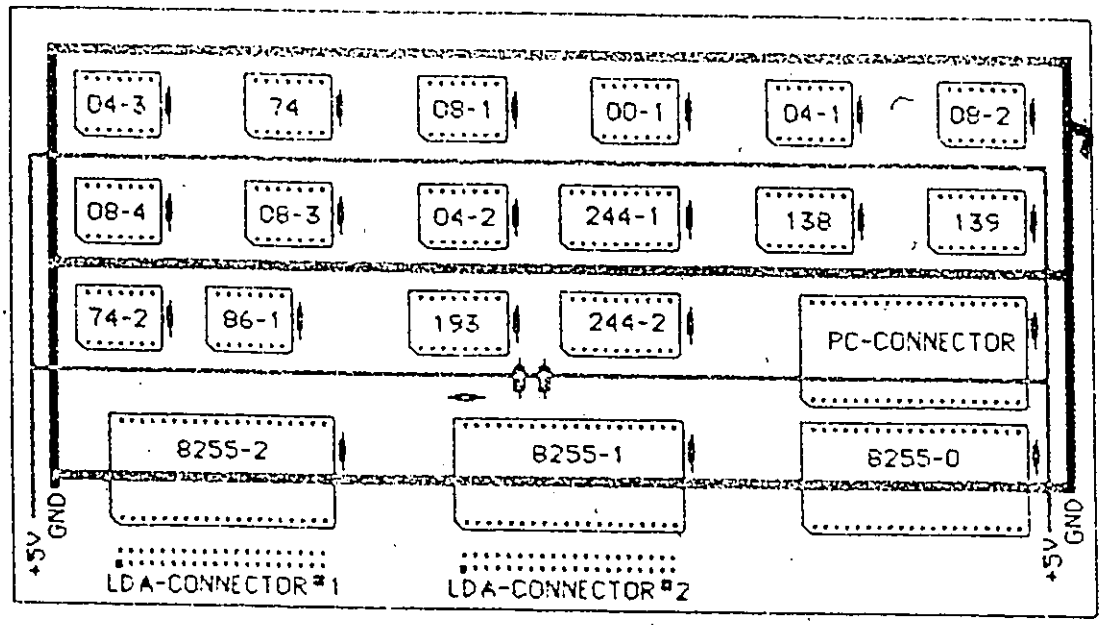
Revision:

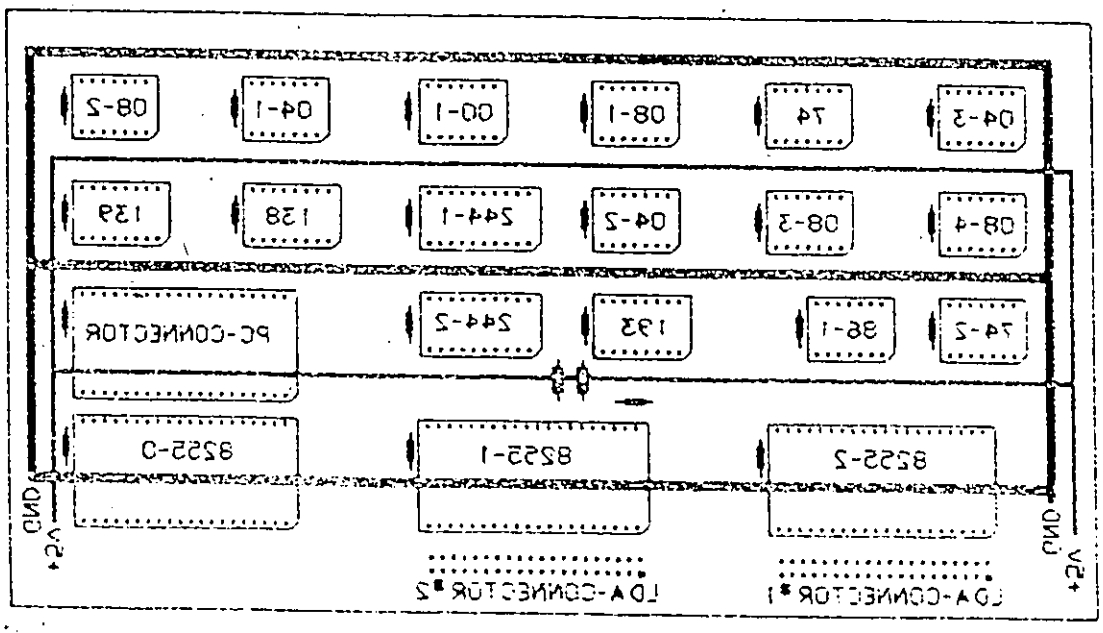
Bill Of Materials October 12, 1988 18:05:08
Page 1

Item	Quantity	Reference	Part
1	2	U244-1,U244-2	74LS244
2	1	U138	74LS138
3	4	U08-2,U08-1,U08-3,U08-4	74LS08
4	3	U8255-1,U8255-0,U8255-2	8255
5	2	U139,U139-1	74LS139
6	5	U04-1,U04-2,U04-3,U-4-02, U04-02	74LS04
7	1	U86-1	74LS86
8	3	U74-3,U74-1,U74-2	74LS74
9	1	SW-1	SW SPDT
10	1	U00-1	74LS00
11	1	U193	74LS193
12	4	C4,C1,C2,C3	1 mf
13	1	R1	2.2K
14	1	R2	1.0K
15	1	R3	330



PC-LDA to Buffer Interface Cable Configuration





PC Board IC Layout

VITA AUCTORIS

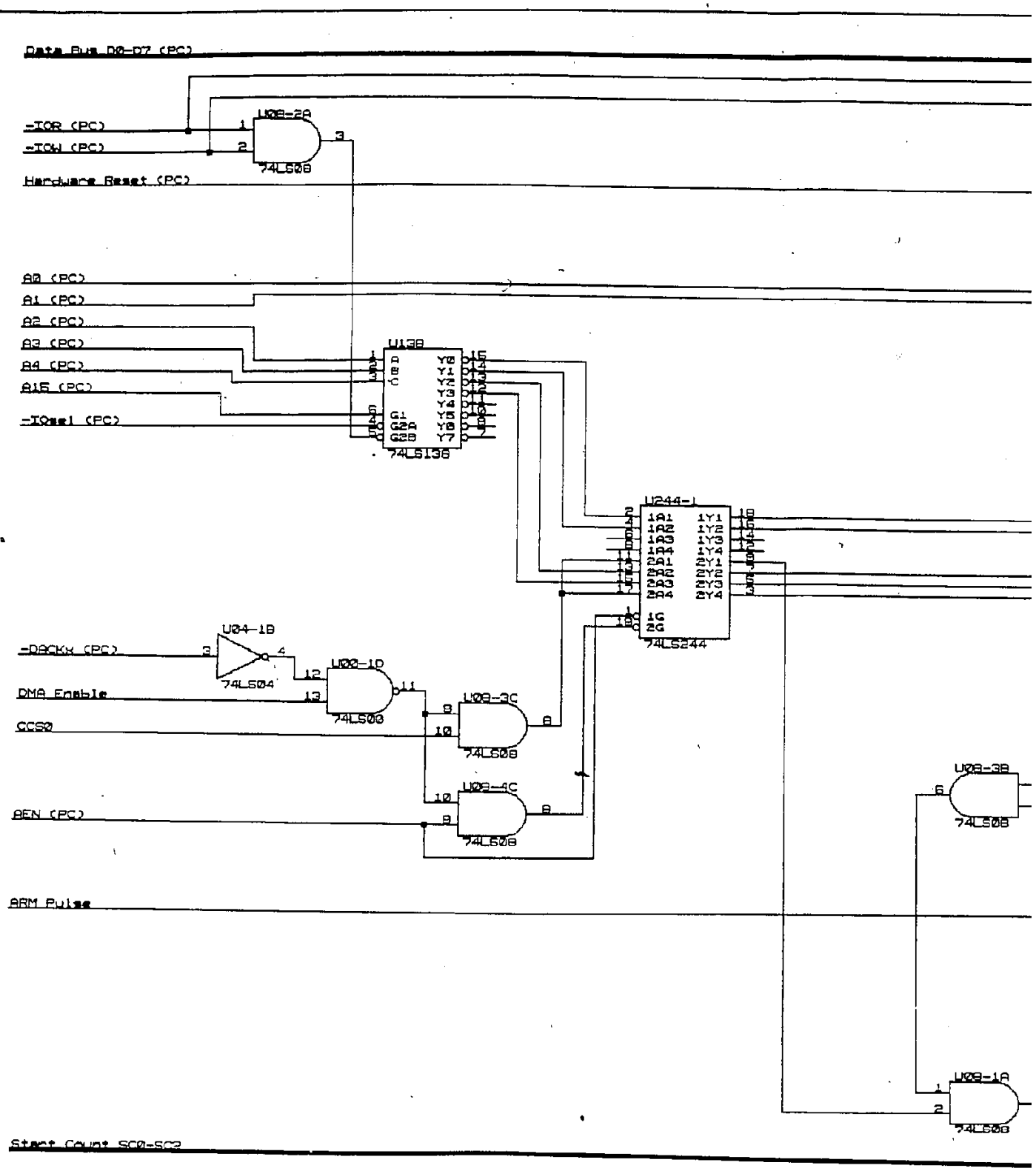
1960 Born in Hong Kong on September 17

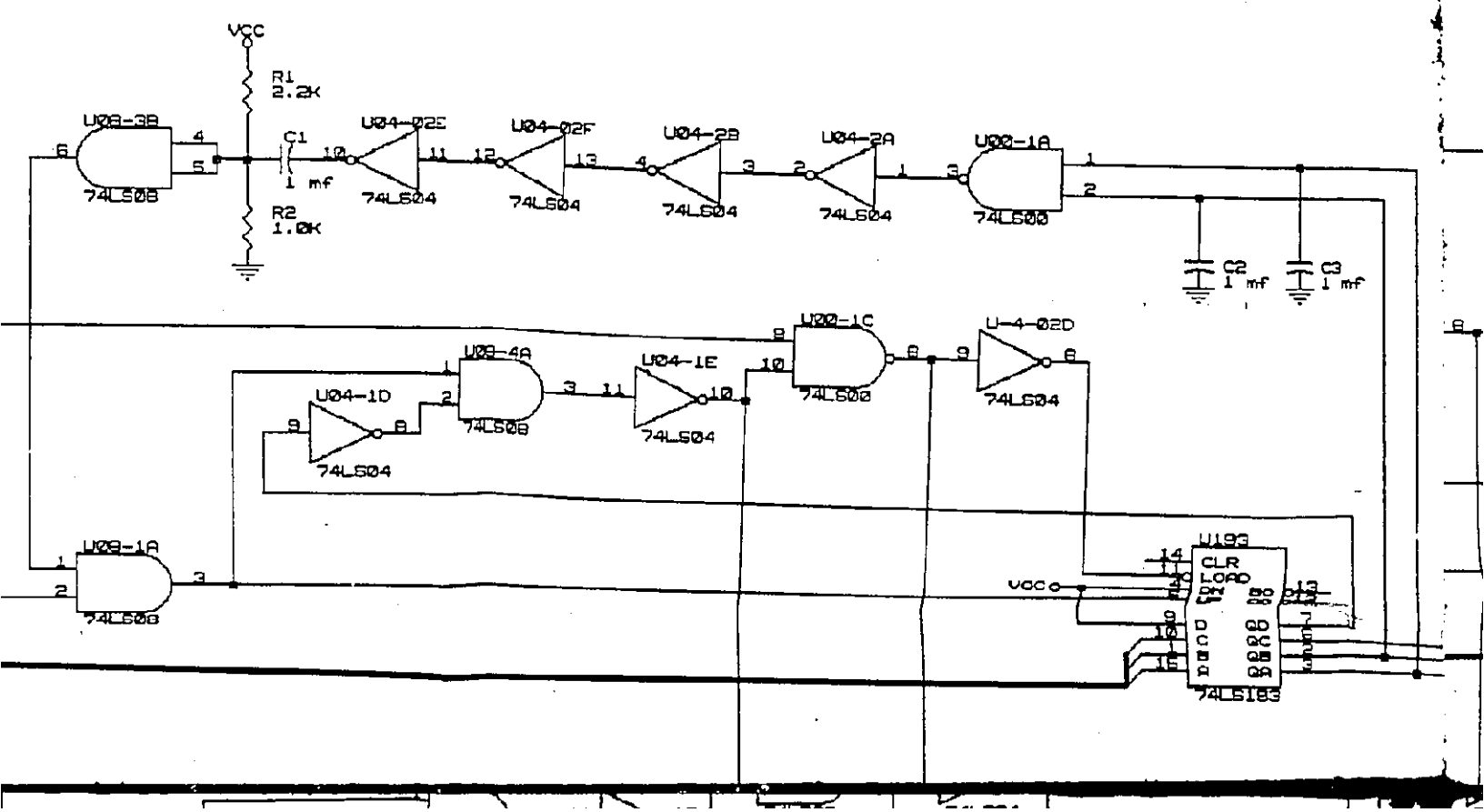
1979 Completed Secondary Education at Forster Secondary School, Windsor, Ontario, Canada in July.

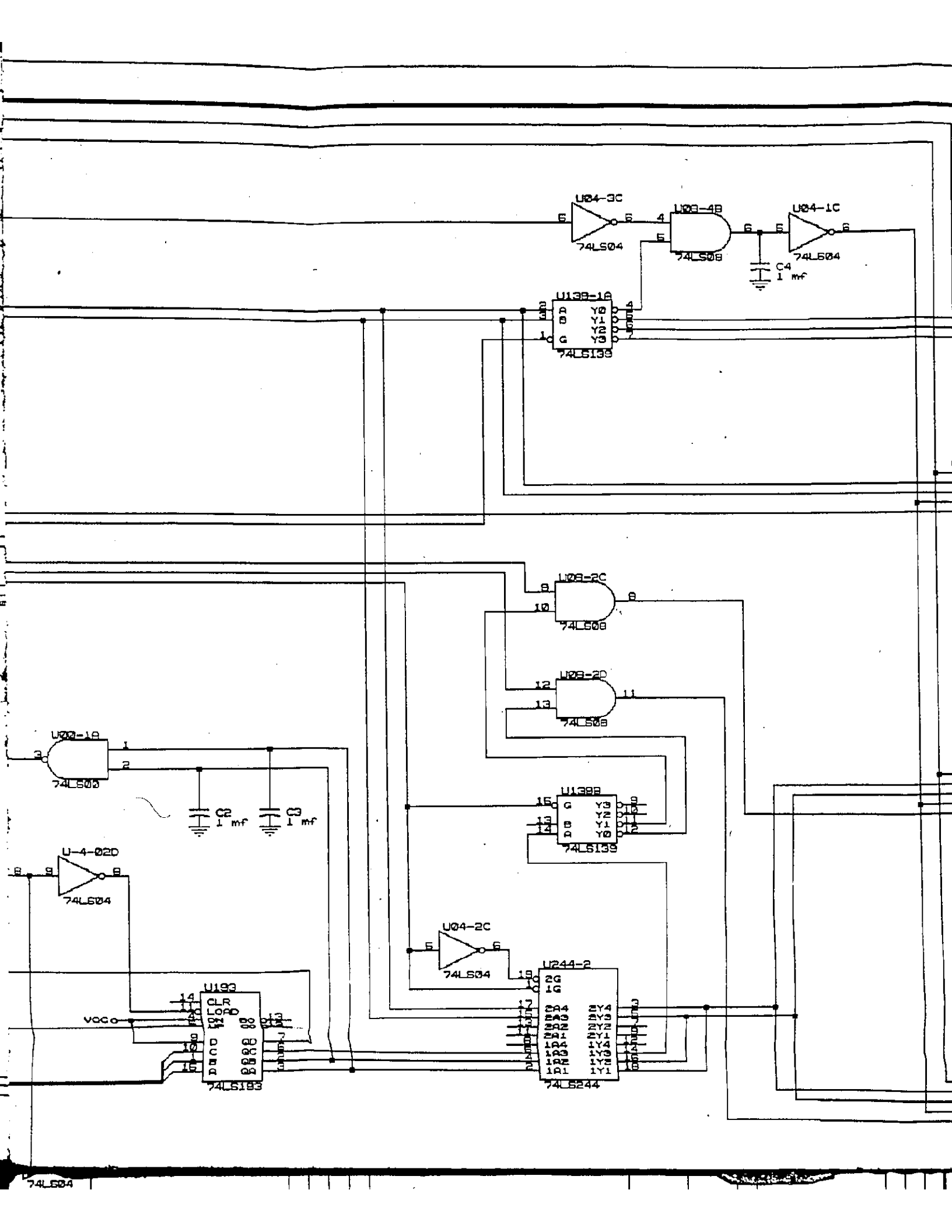
1983 Received the degree of Bachelor of Applied Science in Mechanical Engineering, University of Windsor, Windsor, Ontario, Canada in June.

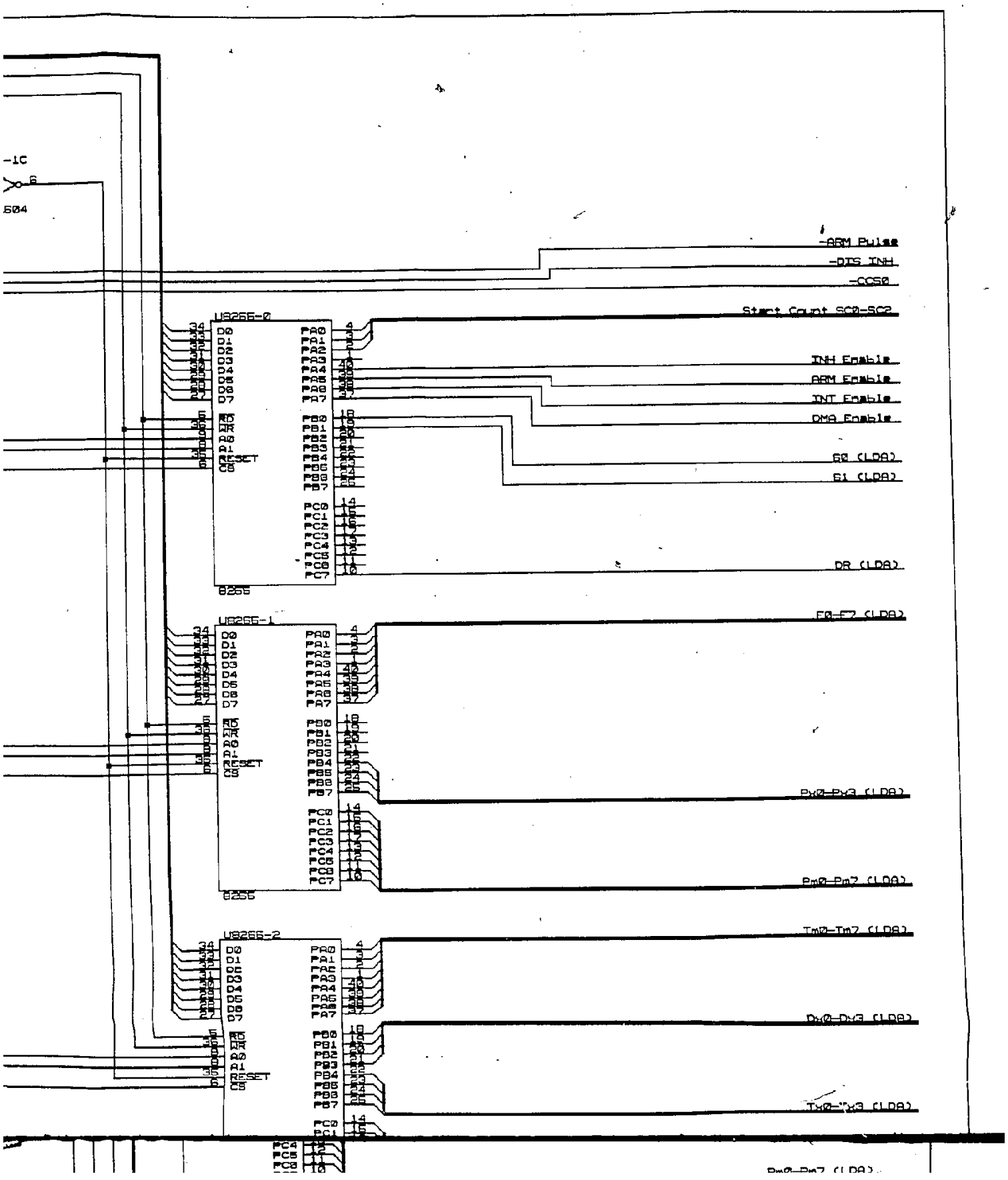
1984 Received the degree of Bachelor of Computer Science, University of Windsor, Windsor, Ontario, Canada in June.

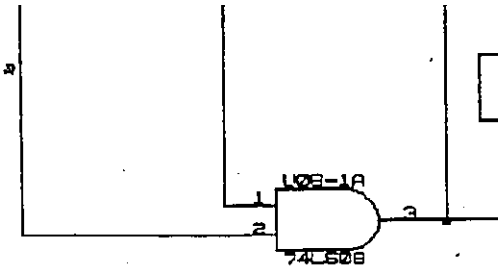
1988 Currently a candidate for the degree of Master of Applied Science in Mechanical Engineering at the University of Windsor, Windsor, Ontario, Canada


Data Bus D0-D7 (PC)
-IOR (PC)
-IOW (PC)
U08-2A
74LS08
Hardware Reset (PC)
A0 (PC)
A1 (PC)
A2 (PC)
A3 (PC)
A4 (PC)
A15 (PC)
-IOsel (PC)
U13B
74LS138
U244-1
74LS244
-DACKx (PC)
U04-1B
74LS04
U00-1D
74LS00
DMA Enable
CCS0
U08-3C
74LS08
U08-4C
74LS08
AEN (PC)
U08-3B
74LS08
ARM Pulse
U08-1A
74LS08
Start Count SC0-SC2


VCC
R1
2.2K
C1
1 mf
R2
1.0K
U00-3B
74LS08
U04-02E
U04-02F
U04-2B
U04-2A
U00-1A
74LS04
74LS00
C2
1 mf
C3
1 mf
U00-1C
U-4-02D
U09-4A
U04-1E
U04-1D
U09-1A
U193
CLR
LOAD
74LS193


U04-3C
74LS04
U08-4B
74LS08
U04-1C
74LS04
C4
1 mf
U138-1B
74LS139
U08-2C
74LS08
U08-2D
74LS08
U00-1B
74LS00
C2
1 mf
C3
1 mf
U139B
74LS139
U-4-02D
74LS04
U04-2C
74LS04
U193
74LS193
VCC
U244-2
74LS244


-1C
504
-ARM Pulse
-DIS INH
-CCS0
Start Count SC0-SC2
U8255-0
INH Enable
ARM Enable
INT Enable
DMA Enable
G0 (LDA)
G1 (LDA)
DR (LDA)
8255
U8255-1
F0-F7 (LDA)
Px0-Px3 (LDA)
Pm0-Pm7 (LDA)
8255
U8255-2
Tm0-Tm7 (LDA)
Dw0-Dw3 (LDA)
Tw0-Tw3 (LDA)
Pm0-Pm7 (LDA)



eset Count SC0-SC2

RM Enable

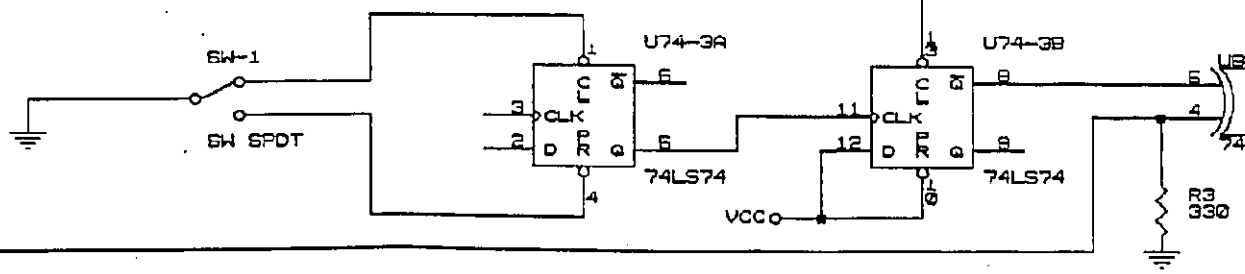
DIS INH

NH Reset

NH Enable

NT Enable

BACKx (PC)

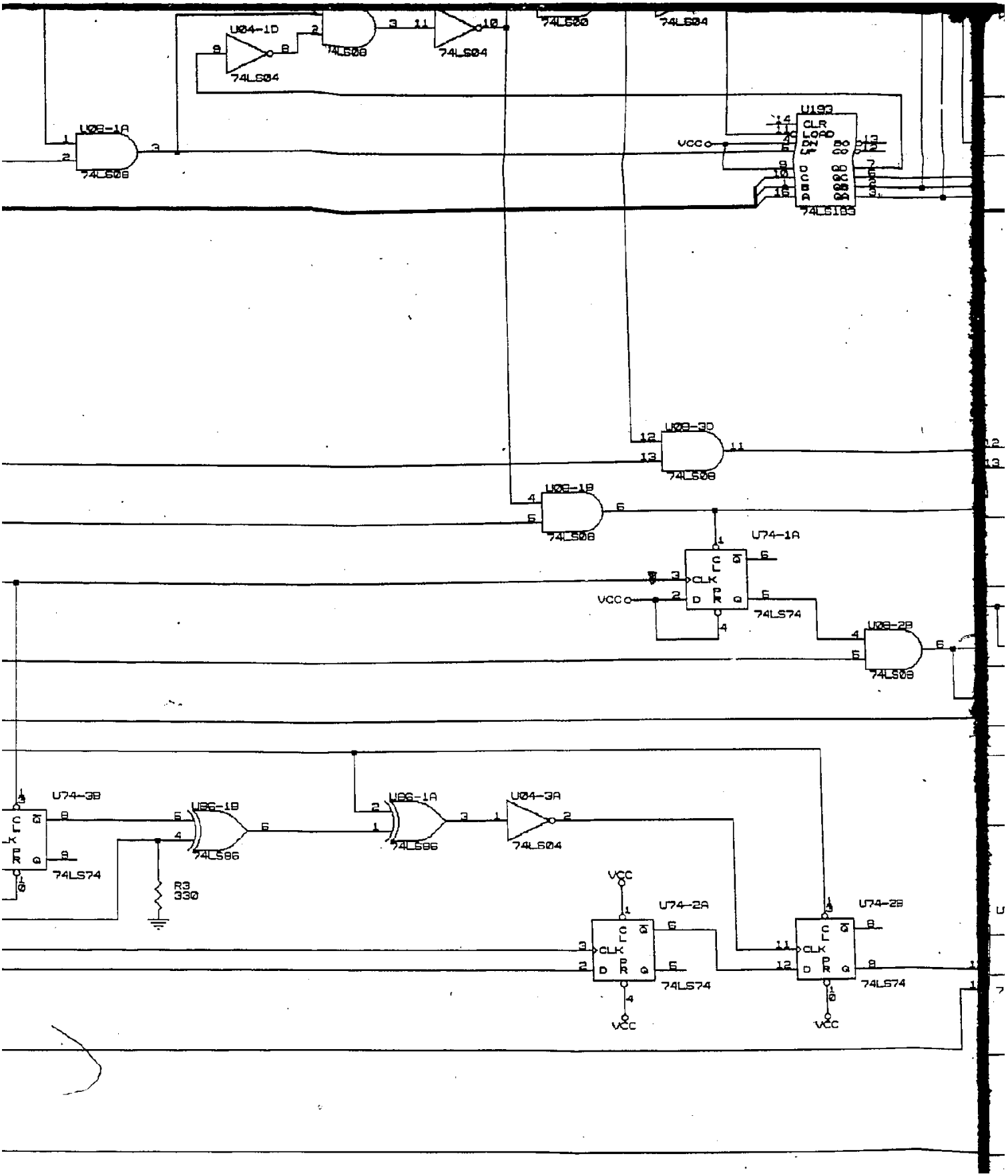


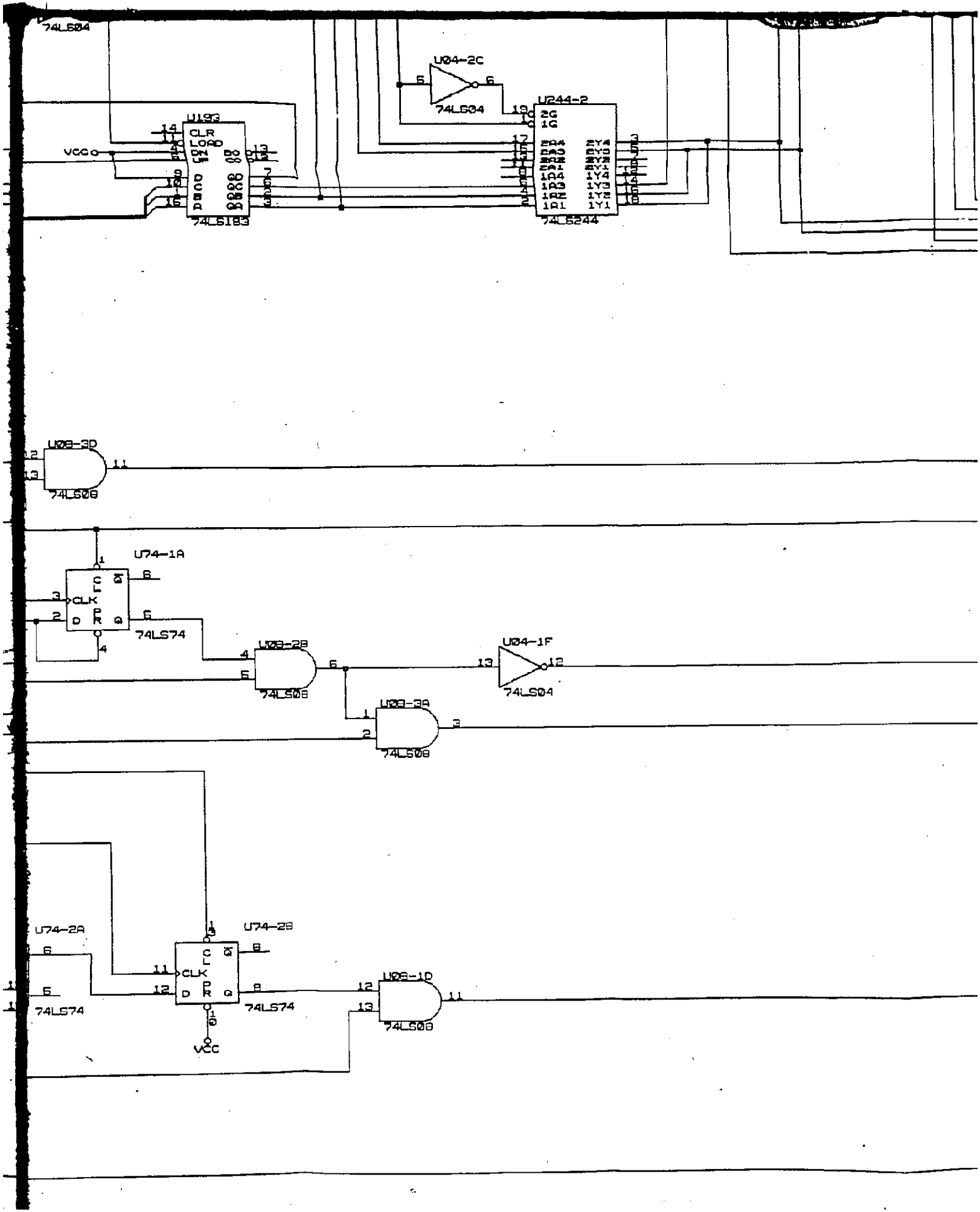
R (LDA)

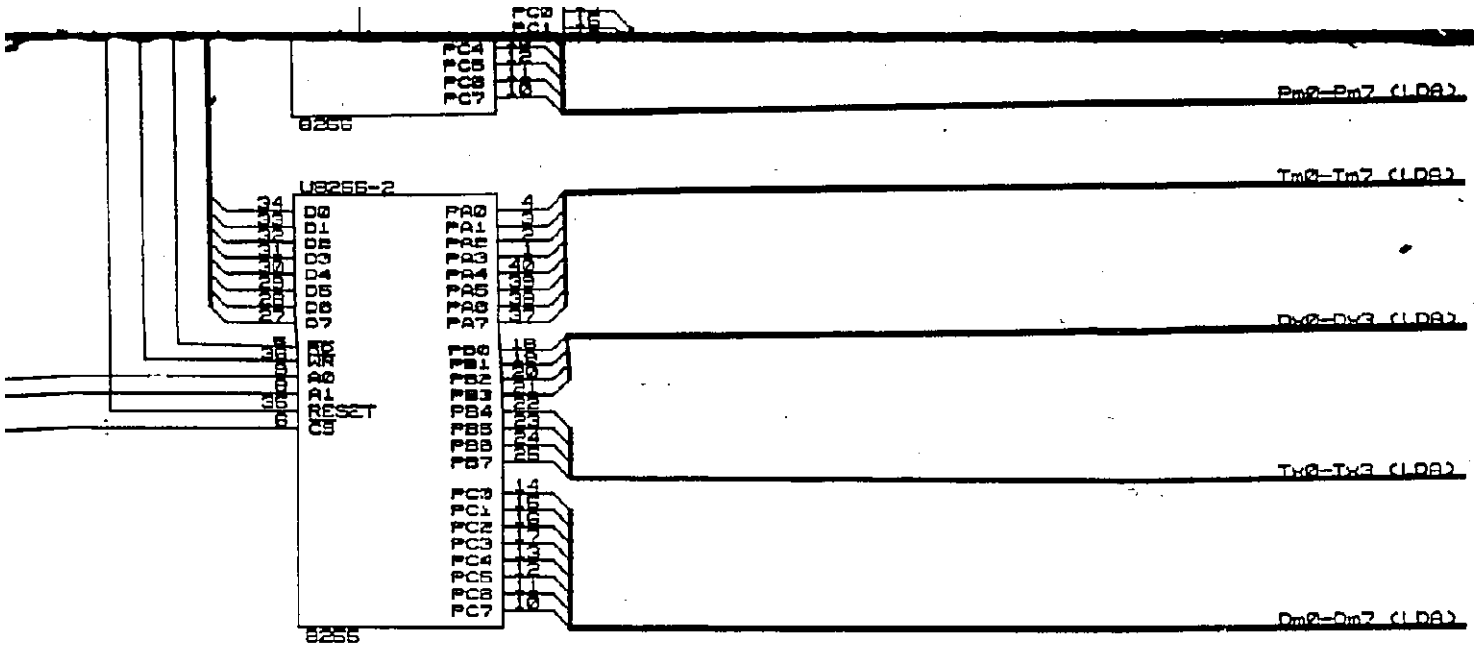
lock (PC)

IC (PC)

MA Enable


U04-1D
74LS04
74LS08
74LS04
74LS00
74LS04
U08-1A
74LS08
U193
CLR
LOAD
VCC
74LS193
U08-3D
74LS08
U08-1B
74LS08
U74-1A
CLK
D
74LS74
VCC
U08-2B
74LS08
U74-3B
74LS74
U86-1B
74LS86
U86-1A
74LS86
U04-3A
74LS04
R3
330
VCC
U74-2A
74LS74
VCC
U74-2B
74LS74
VCC


74LS04
U04-2C
74LS04
U244-2
2G
1G
74LS244
U193
CLR
LOAD
74LS193
VCC
U08-3D
74LS08
U74-1A
CLK
74LS74
U08-2B
74LS08
U04-1F
74LS04
U08-3A
74LS08
U74-2A
74LS74
U74-2B
74LS74
VCC
U08-1D
74LS08



Title	PC-LDA Interface	
Size D	Document Number	REV
Date: October 12, 1988	Sheet 1 of 2	